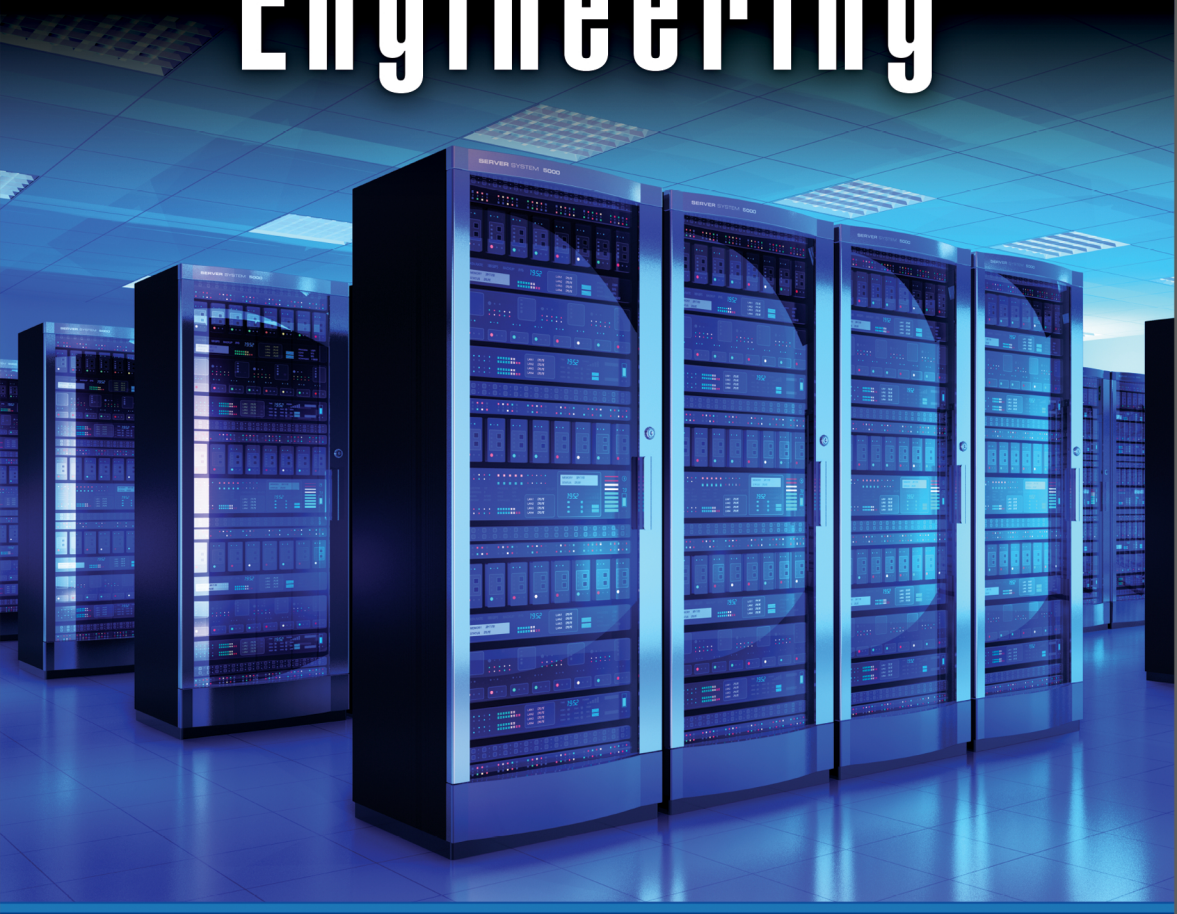
Second Edition

Communication Protocol Engineering



Miroslav Popovic

CRC Press
Taylor & Francis Group

Communication Protocol Engineering

Second Edition

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Communication Protocol Engineering

Second Edition

Miroslav Popovic

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-1-138-55812-0 (Hardback)
International Standard Book Number-13: 978-1-315-15124-3 (eBook)

Library of Congress Cataloging-in-Publication Data

Names: Popovic, Miroslav, 1961- author.
Title: Communication protocol engineering / Miroslav Popovic.
Description: Second edition. | Boca Raton : Taylor & Francis, CRC Press, 2018.
Identifiers: LCCN 2017043058| ISBN 9781138558120 (hardback : alk. paper) |
ISBN 9781315151243 (ebook)
Subjects: LCSH: Computer network protocols. | Computer networks--Standards.
Classification: LCC TK5101.55 .P67 2006 | DDC 621.382/12--dc23
LC record available at https://lccn.loc.gov/2017043058

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To my wife, Vlasta, and our sons Marko and Andrej

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

Preface to the First Edition

I wrote this book as a textbook for postgraduate students, but it might also be used by people in the industry to update specific knowledge in their lifelong learning processes. The book partly covers the actual postgraduate course on computer communications and networks undertaken during the first semester of studies for the M.Sc. degree in computer engineering. Since nowadays we are witnessing the convergence of the Internet and the public telephone network, this book might also be useful to engineers with B.Sc. degrees in telecommunications.

The prerequisite for this book is knowledge of first order logic (predicate calculus), operating systems, and computer network fundamentals. The reader should also be familiar with C++ and Java programming languages.

My approach in writing this book was to provide all the details that the reader may need. I assumed that nothing is obvious. However, if you, the reader, find something obvious while reading the book, you are encouraged to skip ahead. If something is not clear later on, you may always return to what you skipped. Communication protocol engineering is a very interesting combination of abstraction and practice that requires a lot of details. It starts from a vision that gradually materializes in real-world artifacts. This happens through a typical engineering process. This book covers all aspects of communication protocol engineering, including requirements and analysis, design, implementation, and test and verification.

Many people helped me in writing this book. My gratitude goes to all of them. I thank my family for their continuous support; my niece Silvia Likavec for her valuable text corrections; and B.J. Clark, Nora Konopka, and Helena Redshaw, of Taylor & Francis, for their professional support. Special thanks go to my colleagues from the University of Novi Sad; Prof. Vladimir Kovacevic for giving his blessing for this book; Ph.D. student Ivan Velikic for the excellent cooperation (in his M.Sc. thesis we actually developed the FSM Library, one of the anchors of this book); Ph.D. student Ilija Basicevic (for helping me with the preparation of the examples in Sections 3.10.5, 4.5.2, and 5.5.2); Sonja Vukobrat (for helping me with the preparation of the example in Section 3.7); Laslo Benarik and Aleksander Stojicevic (for helping me with the preparation of Chapter 6); Milan Savic; Aleksander Stojicevic; and Cedomir Rebic (for helping me with the preparation of the examples in Sections 3.10.1 and 3.10.2); and Nenad Cetic (for helping me with the preparation of the example in Section 4.5.1). Thank you all!

Miroslav Popovic
Novi Sad

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Preface to the Second Edition

The first edition of this book was well accepted by the readers, right from the beginning, back in 2006 when it was printed. Barnes & Noble bestselling rating reports indicated this fact rather well, e.g., the book was the bestseller on Oct. 4th, 2006, in the section "Networking, Telecommunications Protocols, & Standards." From that time to today, *Communication Protocol Engineering* has been a subject on a number of graduate level (M.Sc.) courses at universities worldwide—from the United States (The City College of New York, New York; University Heights Newark, New Jersey; etc.), over Europe (University of Novi Sad, Serbia; Lippe and Hoexter University of Applied Sciences, Germany), to far-east Australia (La Trobe University, Australia), to name just few of the more established points. Nowadays, *Communication Protocol Engineering* sounds like evergreen, similar to its much older predecessors Internet, C, and Linux, which are with us from the 1970s, and it seems that *Communication Protocol Engineering* is here to stay for many years to come, similar to its famous predecessors.

Twelve years after I wrote the first edition, I was glad to see that it was still aligned with the state of the art very well. Still, the book needed to be improved in two important areas, namely, compliance testing based on the standard Testing and Test Control Notation (known as TTCN-3), and model checking based on famous C.A.R. Hoare's process algebra Communicating Sequential Processes (CSP) and its accompanying tool named Process Analysis Toolkit (PAT). Hence, I made this new edition.

Technically, I made appropriate changes in Chapters 3 and 5. In Chapter 3, I have rewritten Sections 3.9 and 3.10 (Examples 1 and 2), and I adapted the TTCN references throughout the book in order to introduce the current standard TTCN-3 instead of the previous standard TTCN-2 (this decision was driven by the fact that TTCN-3 is a superset of the TTCN-2).

In Chapter 5, I revised Section 5.3. The new title of Section 5.3 is "Formal Verification," and it comprises the following two subsections: (i) 5.3.1. Formal Verification Based on Theorem Proving (this is the original Section 5.3), and (ii) 5.3.2 Formal Verification Based on Communicating Sequential Processes (this is the new section based on C.A.R. Hoare's process algebra CSP and the accompanying modeling, simulation, and automatic verification tool PAT).

Many people assisted me during the writing of this second edition. My gratitude goes to all of them. I thank Nora Konopka and Kyra Lindholm, of Taylor & Francis, for their professional support. Thanks again to my

family, and my colleagues from the University of Novi Sad for their support throughout all these years.

I would also like to express my special gratitude to Dr. Sun Jun and the PAT Team for providing their PAT Examples in the public domain. I used some of their CSP# models to create the examples in Section 5.3.2.3.

Miroslav Popovic
Novi Sad

Author

Miroslav Popovic, Ph.D., earned all his degrees from the University of Novi Sad, Serbia. He defended his diploma thesis, "An Intelligent System Restart," in 1984; his M.Sc. thesis, "An Efficient Virtual Machine System," in 1988; and his Ph.D. thesis, "A Contribution to Standardization of ISO OSI Presentation Layer," in 1990. He became a full-time professor at the University of Novi Sad in 2002. Currently, he is teaching courses on software tools and real-time systems programming, as well as on intercomputer communications and computer networks. He is a member of IEEE (both the Computer and the Communications Societies) and ACM. He has published approximately 120 papers, and he has supervised many real-world projects for the industry, including telephone exchanges and call centers for Russian, German, Czech, and Serbian telecommunication networks. Taylor & Francis published his book, *Communication Protocol Engineering*, in 2006. He served as Serbian MC Member in EU COST 297 High Altitude Platforms of wireless communications, EU COST IC0703 Traffic Monitoring and Analysis, and EU COST Action IC1001 Transactional Memories (Euro-TM). His current research interests are engineering of computer-based systems, parallel programming, distributed systems, and security.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1

Introduction

Originally, the term **protocol** was related to the customs and regulations dealing with diplomatic formality, precedence, and etiquette. A protocol is actually the original draft, minutes, or record from which a document, especially a treaty, is prepared, e.g., an agreement between states. Today, in the context of computer networks, the term **protocol** is interpreted as a set of rules governing the format of messages that are exchanged between computers. Sometimes, especially if we want to be more specific, we use the term **communication protocol** instead.

The title of this book, *Communication Protocol Engineering*, is used to emphasize the process of developing communication protocols. Like other engineering disciplines, communication protocol engineering typically comprises the following phases (Figure 1.1):

- Requirements and analysis
- Design
- Implementation
- Test and verification

The process described in this book is the union of the UML (Unified Modeling Language)–driven unified development process (Booch et al., 1998) and, Cleanroom engineering (formal system design verification and statistical usage testing), with some elements of Agile programming (particularly unit testing based on JUnit). Of course, each organization should adapt and tune the process to its own needs and goals. For example, one organization may stick to the UML-driven unified development process, another may prefer Cleanroom engineering, yet another may use the combination of both, and so forth.

Because this book is written for the process in which all the existing state-of-the-art methods and techniques in the area are applied, it is independent of any particular engineering process. Therefore, this is as far as we will go in discussions on processes in this book. This book is not about managing processes. Rather, this book is intended for engineers. It provides the knowledge that an engineer needs to work in a modern organization involved in communication protocol engineering.

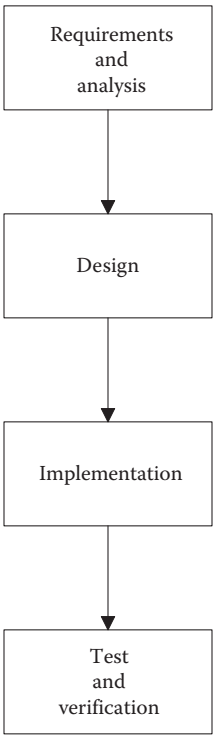


FIGURE 1.1
Typical communication protocol engineering phases.

The chapters are named by typical process phases: requirements and analysis, design, implementation, and test and verification. These chapters are actually used to classify various methods and techniques, and their accompanying tools. As previously stated, the approach taken in this book was to select the best methods and techniques from various methodologies rather than to stick just to a single methodology. The methods and techniques introduced here originate from the following methodologies:

- UML methodology
- ITU-T system specification and description methodology
- Agile unit testing methodology
- Cleanroom engineering methodology

UML methodology is based on various kinds of graphs, also referred to as diagrams. This book covers all of them, namely:

- Use case diagrams (Section 2.1)
- Collaboration diagrams (Section 2.2)

- Class diagrams (Section 3.1)
- Object diagrams (Section 3.2)
- Sequence diagrams (Section 3.3)
- Activity diagrams (Section 3.4)
- Statechart diagrams (Section 3.5)
- Deployment diagrams (Section 3.6)
- Component diagrams (Section 4.1)

ITU-T system specification and description methodology is based on three domain-specific languages, which this book also covers. These languages are

- Specification and description language (SDL) (Section 3.7)
- Message sequence charts (MSC) (Section 3.8)
- Testing and test control notation, ver. 3 (TTCN-3) (Section 3.9)

Agile unit testing methodology assumes writing the test cases before the code. Today, it is supported by the following two open-source packages (both covered in this book):

- JUnit, a package for automated unit testing of Java packages (Section 5.1)
- CppUnit, a library for automated unit testing of C++ modules (Section 5.5.1)

Cleanroom engineering methodology is based heavily on two main methods, both covered in this book. These methods are

- Formal system design verification. Today, more approaches exist to formal system design verification. This book covers formal verification based on automated theorem proving (Section 5.3).
- Statistical usage testing (Section 5.4).

The text of the book is organized as follows. At the end of this chapter, in Section 1.1, we introduce the notion of the communication protocol and related definitions.

Chapter 2 is devoted to the requirements and analysis phase of communication protocol engineering. The first part of that chapter introduces UML use case and collaboration diagrams (Section 2.1 and Section 2.2, respectively). The former is used for capturing both functional and nonfunctional system requirements, whereas the latter is used for making system analysis models. The second part of that chapter presents a real-world example—requirements

and analysis of an SIP (Session Initiation Protocol, RFC 3261) Softphone. The example starts with the presentation of the domain-specific information related to SIP, continues with the SIP Softphone requirements model (in the form of the corresponding use case diagram), and ends with the SIP Softphone analysis model (in the form of the corresponding collaboration diagram).

Chapter 3 covers the design phase of communication protocol engineering. In this chapter, we will see that communication protocols are actually modeled as finite state machines (FSMs). The first part of the chapter introduces UML diagrams related to the design phase: class, object, sequence, activity, statechart, and deployment diagrams (Section 3.1, Section 3.2, Section 3.3, Section 3.4, Section 3.5, and Section 3.6, respectively). The second part of Chapter 3 covers domain-specific languages, which originated at ITU-T, namely SDL, MSC, and TTCN-3 (Section 3.7, Section 3.8, and Section 3.9, respectively). The third part consists of design examples. The first three examples are rather academic, while the fourth example shows the design of the sliding window concept. The fifth example is a real-world design example—the design of the SIP INVITE client transaction, which is a part of the SIP protocol stack.

Chapter 4 is devoted to the implementation phase of communication protocol engineering. At the beginning of this chapter, we introduce the UML component diagrams (Section 4.1). The second part of Chapter 4 presents various implementation approaches. Section 4.2 presents three examples of approaches that can be used. The main goal of this study is to provoke dilemmas by studying three different concepts of implementation and to promote creative thinking about a spectrum of possible implementation paradigms before restricting ourselves to a single one. This short overview includes the implementations as nested switch-case statements, the implementation based on the interpretation of protocol messages using a protocol definition data structure, and the implementation based on a class hierarchy and state transition map. The second part of Chapter 4 ends with the introduction of the state design pattern (Section 4.3), with a catalogued FSM implementation approach.

The third part of Chapter 4 (Section 4.4) introduces one concrete, industrial-strength implementation paradigm based on the FSM Library, a library of C++ classes used for modeling communication protocols as FSM. This paradigm has been successfully used on a series of real-world projects, such as SS7, DSS1, V5.2, H.323, SIP, and so on. This part of the book covers FSM Library features and internals as well as the rules for writing FSM Library–based implementations. The last part of Chapter 4 contains two real-world examples of the FSM Library–based implementations. The first is the implementation of the POP3 communication protocol, the TCP/IP Internet protocol for receiving e-mail messages. The second is the SIP INVITE client transaction, a part of the SIP protocol stack.

Chapter 5 deals with the testing and verification phase of communication protocol engineering. The first part starts with the introduction of unit testing based on JUnit, the open-source testing framework for unit testing Java programs, originally developed by Erich Gamma and Kent Beck (Section 5.1). Next, we introduce conformance testing (Section 5.2), actually the first stage of communication protocol acceptance testing. Conformance testing is typically based on the TTCN test suite specification. We then introduce formal verification of both system design and implementation (Section 5.3) based on: (i) automated theorem proving (Section 5.3.1) and (ii) the C.A.R Hoare's process algebra CSP (Section 5.3.2). In this book, we use the theorem prover Theo (in Section 5.3.1) and the modeling, simulation, and automatic verification tool PAT (in Section 5.3.2) as the accompanying tools for this purpose.

The first part of Chapter 5 ends with the introduction of statistical usage testing (Section 5.4) based on product operational profiles. The second part of Chapter 5 consists of two real-world examples. The first example shows the unit testing of the SIP INVITE client transaction based on the usage of the CppUint, the library for unit testing C++ modules. The second example demonstrates the integration testing of the SIP INVITE client transaction.

Chapter 6 is written as a programmer's reference manual for the FSM Library. The first part starts with the introduction of two main classes, *FSMSystem* and *FiniteStateMachine* (Section 6.2). Next, we introduce three main groups of basic functions supported by the FSM Library: time, memory, and message management functions (Section 6.3, Section 6.4, and Section 6.5, respectively). We then introduce two classes that support the communication of FSMs over the TCP/IP Internet (Section 6.6), namely the classes *FSMSystemWithTCP* and *NetFSM*. The first part of Chapter 6 ends with the introduction of global constants, types, and functions (Section 6.7).

The second part of Chapter 6 contains detailed descriptions of the individual FSM Library Application Programming Interface (API) functions (Section 6.8). The third part of Chapter 6 consists of two examples. The first is a simple example with three automata (FSM) instances (Section 6.9), and the second is a simple example with TCP/IP network-aware automata instances (Section 6.10).

1.1 The Notion of the Communication Protocol

What is a communication protocol? A wide range of definitions are available in the literature today, for example: "An established set of conventions by which two computers or communication devices validate the format and content of the messages exchanged;" "A set of defined interfaces that permits the computers to communicate with each other;" "A method by which two computers coordinate their communication;" "Common agreed rules

followed in order to interconnect and communicate between computers;" "The rules governing the exchange of information between devices on a data link;" "The set of rules governing how information is exchanged on a network;" and so on.

In this book, we begin with a wider informal definition. A **protocol** is a set of conventions and rules governing their use that regulates the communication of an entity under observation with its environment. Such a definition enables the study of any communication, e.g., an agenda for a technical meeting of representatives of two companies. The subject of this book is one special class of protocols, referred to as **communication protocols**, that regulate the communication of geographically distributed program objects. The communicating program objects are deployed on different processors in the network. We will sometimes use the term **protocol** as an abbreviated form of the phrase **communication protocol** to save space.

A **process**, as generally defined in the theory of operating systems, is a program in execution or prepared for execution. A process may be specialized for data processing, communication, or some other special task (e.g., I/O control or time management). Traditionally, a data processing algorithm is specified by the flowchart. What the flowchart means for the data processing process, the protocol means for the communication process.

The flowchart specifies the program control flow by the use of graphic symbols related to the series of sequential calculations, selection, iteration, procedure/function call, and input/output operations needed to read input data or write output data. On the other hand, the formal specification of a communication protocol is based on messages and consists of the following three parts:

- The message format specification
- The message-processing procedures specification, which is essentially a formal description of process reactions to input stimuli (i.e., messages)
- The error processing specification, which is the formal description of process reactions to exceptional events (i.e., corrupted data or timeouts)

The **message format** completely defines the structure of the message, i.e., it defines the set of fields that constitute the message by defining the width of individual fields (most commonly in bits, bytes, or words), the applied coding scheme (e.g., binary, ASCII, Unicode, ASN.1), and optionally legal values (e.g., constants in binary or some symbolic form or value intervals).

Therefore, a **message** is a series of bits logically divided into various fields. Typically, a message consists of a message header, which most commonly comprises more subfields, and useful data referred to as a payload. The **payload** contains data interpreted by the communicating program objects.

The message header contains data added for supervision and control purposes in accordance with the established conventions.

The **message-processing procedure** (i.e., the process reaction) begins with the message reception and is described as a series of primitive operations that define the rules of the communication, which are the essential parts of a protocol. Typical **primitive operations** include timer-start operations, timer-stop operations, message-send operations, message-receive operations, and message-data processing operations (e.g., cyclic redundancy checking of message data, calculating the expected order number of the next message to be received).

In terms of software implementation, message processing is performed by a message processing routine. Depending on the selected working environment, this routine can be a subroutine that consists of a series of machine instructions in a symbolic form (assembly language) or a function comprising a series of statements in a higher-level programming language, such as C/C++ or Java.

The error-processing specification defines a set of error reactions. An **error reaction** is a special protocol reaction to exceptional events or, in other words, a reaction to unexpected situations, i.e., conditions. Typical examples of unexpected events are the reception of a message that contains corrupted data, the reception of a message that is out of the original order (e.g., after receiving the messages numbered 1, 2, and 3, we receive the message numbered 7 instead of the message numbered 4), timer expiration (e.g., the receiver has not acknowledged the reception of a message to its sender within a certain interval of time, determined by the value of the corresponding timer), and so on.

Note that a protocol can be described informally or formally. The informal description of a protocol is referred to as its **informal specification** and has the following characteristics:

- It frequently has the form of a combination of textual and graphical descriptions of the most common scenarios of communication.
- It may state nothing about the order of the activities to be conducted in the course of the communication.
- It is always incomplete. Most frequently, missing parts are specifications of timers, which determine time limits over individual phases of communication.

Let us forget communication protocols for a moment and use the old example of informal specification of a group of tasks to get a sense of the issues stated above. While leaving the house, a mother says to her daughter:

"Do not forget to finish your homework."

"Have your breakfast when you get hungry."

"Before you go to school, throw the garbage out."

Obviously, this specification does not say anything about the order of the individual tasks. For example, the daughter may complete the tasks in any order without interrupting the individual tasks (e.g., task order may be 1, 2, 3, or 1, 3, 2), or she may complete them in any order and switch between them (e.g., she starts with task 1; then, she switches to task 2 before completing task 1; she completes tasks 2 and 3; and, at the end she finishes task 1). An essential question here is how to organize the task executions within the allocated time. Clearly, a need exists to limit or control the task execution time. What happens if the daughter gets preoccupied with her homework and forgets to have breakfast before it is time to go to school?

The example above might appear to be an exaggeration of the problems we face in reality, but its purpose is to show that informal systems specification alone is insufficient, and that we need a formal systems specification to make a precise and correct system implementation. Formal specification in the area of communication protocols is based on modeling a protocol as a **finite state machine** (FSM). A single FSM is often referred to by the term **automata,** and we will use these two terms interchangeably in this book.

The formal specification of an FSM defines all its states and state transitions, including transitions initiated by expiration of timers, in a unique and detailed way. Today, we may make formal protocol specifications in either UML or ITU-T domain-specific languages. Once we have a formal protocol specification, we can implement it in Java or C++. Finally, we must test and verify it. This procedure is basically what this book is all about.

References

Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1998.

Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1998.

2

Requirements and Analysis

At the beginning of any project, engineers face the fundamental question, "What must be done and how do we verify (deliver) the solution (system, device, products, service, hardware or software)?" Answering this question leads to what are called **requirements**. To simplify the matter, the process of answering this question—i.e., the corresponding engineering phase—is also commonly called *requirements*. Although both the working phase and the resulting documents have the same name, the meaning is easily deduced from the context.

The previous question actually consists of the following two questions:

1. What must be done?
2. How can the solution be verified?

Answering the former question leads to a set of functional requirements, most frequently adorned by nonfunctional requirements. **Functional requirements** describe the desired system behavior, while **nonfunctional requirements** can be imagined as the additional attributes to the behavior related to time restrictions, performance, and so on. To answer the latter question, we must quantify the behavior of the system. Normally, we would say, "For this input, the system should produce this output." Such thinking implies the existence of a test setup that enables automated (most preferably automatic) testing, referred to as a **test bed**. A test bed provides a **test harness** by generating the input to the system and capturing its output.

The ordered pair of the given input and the expected output informally stated in the text above is called a **test case**. To verify complex systems, we need many test cases. A set of test cases packed in a suitable form is referred to as a **test suite**. Ideally, we would like the test suite to completely cover the systems behavior (i.e., the functional requirements), which are adorned with their nonfunctional requirements. Typically, one or more test cases will be derived from each functional requirement. Clearly for any nontrivial system, the number of test cases needed to verify the system may be huge.

However, while thinking about the desired behavior of the system and its verification, we inevitably think about the question, "How can we make it?" Actually, we are trying to make a concept of the system or, more precisely, its architecture. This engineering phase is called an **analysis**. Obviously, it is tightly coupled with the requirements. These two phases have a highly interactive relation.

Typically, work on the definition of the system architecture yields the refinement of system functional requirements, and vice versa. This is especially true for communication protocol engineering. Therefore, we think of these two phases, the requirements and the analysis, as one indivisible front-end phase of communication protocol engineering. This is the reason they are covered together in this chapter.

As previously mentioned, the area of communication protocol engineering is very well founded; many standards, recommendations, and well-known experiences exist—hence, this chapter is rather short compared to the others. Unlike other areas of engineering, a vast majority of engineers here will be faced with the task of implementing some already defined standards, such as IETF RFC or ITU-T/ETSI recommendations, and so on. Very few engineers will be in a position to create a completely new protocol, and even then they will have many existing protocols for reference or starting points.

Many existing standards actually represent very detailed designs accompanied by the corresponding test suites, but others are rather informal, bringing nothing more than the message syntax and encoding together with some textual explanations of the message handling procedures. However, most of the standards can be viewed at least as rather good starting functional requirements that must be further formalized and analyzed. This chapter tries to help the reader exactly in this direction. It tries to answer the question, "How can we deal with the requirements in a systematic way?" Or, in other words, "How do we capture the requirements and how do we proceed with forward engineering from there?"

The overall consensus, in both academia and industry today, is that the UML paradigm can help in this respect (Booch et al. 1998). The behavior of the system is described with a set of use cases. Each **use case** captures one functional requirement adorned with its corresponding nonfunctional requirements. The **requirements engineer** models the system are by specifying the individual actors and the corresponding use cases of the system. The result is referred to as a **requirements model** of the system. The means for making such models are **use case diagrams**, which will be introduced in Section 2.1.

The next step in the UML paradigm is to transform the requirements model into the **analysis model**. Typically, a use case is viewed as a collaboration of classifiers. In the analysis model, three different **stereotypes** of classes are used: *<<boundary class>>*, *<<control class>>*, and *<<entity class>>*. The means of specifying the collaborations in UML are **collaboration diagrams**, which will be introduced in a following section.

Sometimes the analysts describe the static structure of the system—in addition to its behavior—with **class diagrams**. This practice can be helpful in really complex systems. In this chapter, we will present the collaboration diagrams sufficient for the examples at hand, therefore the introduction to class diagrams is postponed until the next chapter. Chapter 3 deals with the communication protocol design phase in which class diagrams are essential to show the static relations among classes.

Further on, in accordance with the UML paradigm, the requirements model should be transformed into the **test model** to facilitate the system verification (the test model is actually the test suite needed for the system verification). Essentially, the use cases should be translated into the corresponding test cases described by test scripts of some kind. UML is not specific in that respect. Of course, a few scripting languages are popular today, such as TCL/TK, Perl, and Payton, but being general purpose languages, these might be inappropriate for some of the projects.

To close this gap, we will introduce a domain-specific language known as **testing and test control notation, ver. 3 (TTCN-3)**. The TTCN-3 language is used for specifying the test suites for communication protocols once the software architecture is rather well known. Therefore, we will postpone the introduction to the TTCN-3 language until Chapter 3, which deals with the design phase of communication protocol engineering.

A general problem when transforming use cases to test cases is that the transformation is typically done manually, i.e., it is semiautomatic. Such an approach is both time consuming and prone to error. However, the main conceptual problem is the test coverage of the system behavior. In practice, the number of possible scenarios and all possible combinations of message parameters can be impossible to cover manually. Therefore, testing at least the most frequently used system scenarios and message parameter combinations should somehow be possible.

Clearly, more detailed UML models made during the system design phase (e.g., statecharts, to be introduced in the next chapter) can be used later for the automatic generation of test cases. However, the problem with this approach is that if an error exists in the UML model, it will be propagated into the test suite and the test suite will not be able to detect the error. A well-known principle from mathematical logic is that negation of negation leads to affirmation, so the bug will remain undiscovered. No matter how large test suite we generate, it will not be able to detect the bug.

The former problem can be solved by the application of **statistical usage testing**, also referred to as **behavior testing**. This paradigm is based on the operational profile model of the system, which describes the statistics of the system usage. It enables the practitioners to thoroughly test the system and even estimate the system or software reliability. This practice is recognized as a *de facto* standard by the industry (Broekman and Notenboom, 2003), and it will be covered in detail in Chapter 5 (test and verification phase of communication protocol engineering).

The latter problem can be solved by using one model as a source for the software implementation generated with forward engineering and a completely different model for the system test suite generation. What is also highly desirable is that these two models are made by two separate individuals or teams. For example, the well-known Cleanroom engineering paradigm is conducted by three completely separate teams. The design team makes the design and does its formal verification, the implementation team just does the coding, and the test team makes the operational profile of the system and conducts the statistical usage testing. Cleanroom engineering will be described together with statistical usage testing in Chapter 5.

Before proceeding further to the introduction of the mainstream approach to requirements and analysis, which is based on UML, it is worth mentioning that, until recently, many opponents to this paradigm existed. Some ongoing doubts still exist as to whether this is the correct choice. For example, in his article, "Use-Cases Are Not Requirements," Meyer argues that a better approach to requirements and analysis is transforming the functional requirements into the behavior model that takes the form of a finite state machine (FSM) (Meyer and Apfelbaum 1999). He sees use cases as just walks across the FSM and claims it is possible to generate them automatically rather than writing them manually.

According to the methodology proposed by Meyer, after creating the behavior model, two parallel streams of activities are started. The first stream covers the analysis, the design, and the implementation, and yields the implementation. The second stream covers the operational profile and the performance analysis, as well as the automatic test suite generation. These two streams merge at the automated testing phase.

This approach is very similar to the one used in this book. A slight difference is that the latter promotes separation of concerns between design and implementation and promotes test teams, including the models they make, very much like the Cleanroom engineering model does. Also, it gives more credit to the UML use cases. If we go back to the original ideas of the UML authors (Booch et al., 1998) and try to think of a single use case as a family of closely related collaborations among the same set of objects, clearly a use case really captures a part of the traditional **list of functional requirements**. Use cases help us group simple and closely related functional requirements, as will be illustrated by the examples in this chapter.

As already mentioned, use cases are the starting point of the software development in the unified software development process (Booch et al., 1998). The requirements model, essentially a set of use cases, is used to develop all the models that correspond to the engineering phases of the process, namely, the analysis model (result of the analysis phase), the design and deployment models (results of the design phase), the implementation model (result of the implementation phase), and the test model (result of the test preparation phase). The focus of this chapter is on requirements and analysis modeling.

The rest of the chapter is organized as follows: the use case and collaboration diagrams are introduced in the next two sections. The last section of this chapter illustrates the requirements and analysis phases of communication protocol engineering by presenting the case of the session initiation protocol (SIP), RFC 3261 (Rosenberg et al., 2002). That last section is divided into three subsections: SIP domain specifics, the SIP requirements model, and the SIP analysis model.

2.1 Use Case Diagrams

Use case diagrams are special kinds of graphs whose vertices are connected with arcs. Two types of vertices are found in use case diagrams, namely, actors and use cases (Figure 2.1). The **actors** represent humans, machines, or software components that are the users of the software under development. They are rendered as stick figures. **Use cases** represent possible uses of the software under development and are rendered as ellipses. As already mentioned, we think of use cases as collaborations between the corresponding objects that constitute the part of the software under development. Clearly, they have different roles in the requirements and the analysis phases.

In the requirements phase, we concentrate on the functional requirements and use the use cases to capture them ("What must be done?"). At that time, how these requirements will be fulfilled does not matter. The only important concern is to build a vision of the future system together with the customer. This vision is expressed as a desirable behavior of the system and modeled by drawing the use case diagram and writing down the descriptions of the individual use cases as they are added to the diagram.

In other words, we concentrate on the client's perspective of the system. The requirements engineer tries to define the services that the system under development should provide. They also try to define an interface to these services. Later, the main problems that the requirements engineer must face are

- Structuring the set of use cases by establishing the relationships among them
- Prioritizing the set of use cases by assigning different priorities to the individual use cases (especially important for the evolving systems)

Use cases have another role in the analysis phase. The job of the analyst is to realize the use cases by the corresponding collaborations between objects. The analyst reads the descriptions of the use cases and uses domain-specific knowledge to identify the individual objects (horizontal structuring) and to

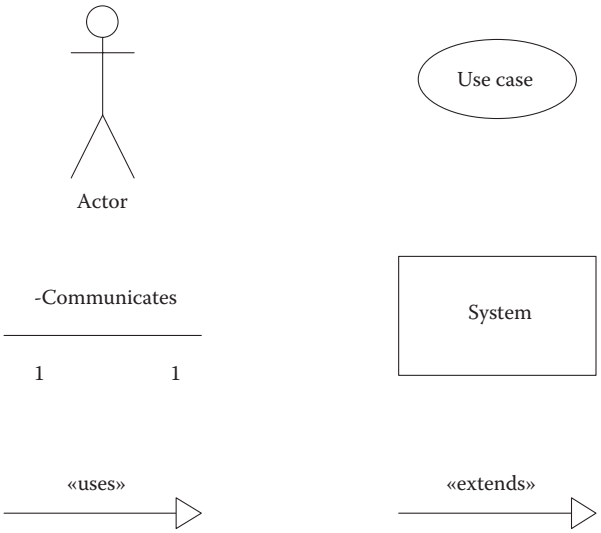


FIGURE 2.1
Basic set of graphical symbols available for rendering use case diagrams.

establish a hierarchy among them (vertical structuring). This process will be described in the next section.

Both actors and use cases are classifiers and, normally, they are connected by associations. The association between the actor and the use case shows the communication between the user and the part of the system modeled by the use case. Using associations enables us to indicate explicitly the points of connection between the users and the system.

Because both actors and use cases are classifiers, we can define general actors and general use cases and then specialize them using the generalization relationship. For example, we may specify the general actor *Client* and its specializations *SIP Client* and *H.323 Client* (Figure 2.2). Or, we can specify the general use case *Make a connection* and its specializations *Make a local connection* and *Make a long distance connection* (Figure 2.3).

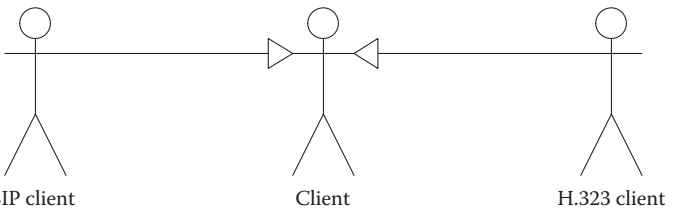


FIGURE 2.2
Example of the generalization and specialization of actors.

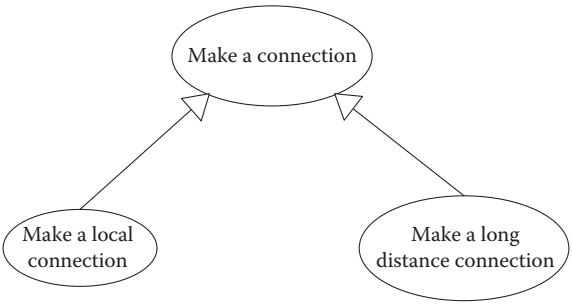


FIGURE 2.3
Example of structuring use cases.

Furthermore, while capturing the individual use cases, it may become obvious that a certain use case extends another use case or that a certain use case includes some other use cases. In such circumstances, the requirements engineer may structure the use cases using <<*extends*>> and <<*includes*>> stereotyped relationships. Especially important things can be indicated by using the sticky notes. Invariants, preconditions, and postconditions can be specified by the corresponding constraints. In more complex use case diagrams, we may need to indicate the packages and the interfaces.

Use case diagrams are normally rendered using the appropriate graphical tools, e.g., Microsoft® Visio. This tool provides the set of graphical symbols that are placed on the working sheet by the drag-and-drop paradigm. The basic set of graphical symbols is shown in Figure 2.1. The requirements engineer must specify the properties for each instance of a symbol in the drawing.

Five categories of actor properties are found: general information, table of attributes, table of operations, table of constraints, and tagged values. The general information includes name, full path, stereotype, visibility (private, protected, or public), and the indicators for *Root*, *Leaf*, and *Abstract* types of actors. The table of attributes includes columns for the attribute name, type, visibility, multiplicity (1, *, 0..1, 0..*, 1..1, or 1..*), and its initial value. The table of operations comprises columns for the operation name, return type, visibility, scope (classifier or instance), and the indicator for the polymorphic operations. The table of constraints consists of four columns: the constraint name, stereotype (precondition, postcondition, or invariant), language type (OCL, text, pseudocode, or code), and body of the constraint. The tagged values include notes for the documentation, location, persistence, responsibility, and semantics.

A use case—being a classifier like an actor—has the same five categories of properties as the actor, as well as the additional sixth category. The sixth category of the use case properties contains the notes about the extension points that are used to describe the <<*extends*>> stereotyped relations.

An association between an actor and a use case has three categories of properties: general information about the association, a table of constraints, and tagged values. The general information includes the association name, full path, stereotype, direction (none, forward, and backward), association end count (default 2), and the attributes for each end of the association. The attributes of the association end are its name, aggregation (none, composite, or shared), visibility, multiplicity, and navigability indicator (navigable or not). The graphical symbol *System* is used to show the system boundaries, i.e., to group the use cases that constitute the system under development. It has no properties.

All the relations between the use cases have three categories of properties: general information, table of constraints, and tagged values. The general information includes the relation name, full path, stereotype (extends, inherits, private, protected, subclass, subtype, or uses), and discriminator. The table of constraints is the same as the table of constraints for the actors and use cases. The tagged values are notes for the documentation.

The additional graphical symbols available for drawing use case diagrams are shown in Figure 2.4. These symbols include notes, general constraints, two-element constraints, OR constraints, packages, and interfaces. The notes have two categories of properties: general properties and tagged values. The general properties include the note name and its stereotype (none or requirement). The tagged values are notes for the documentation.

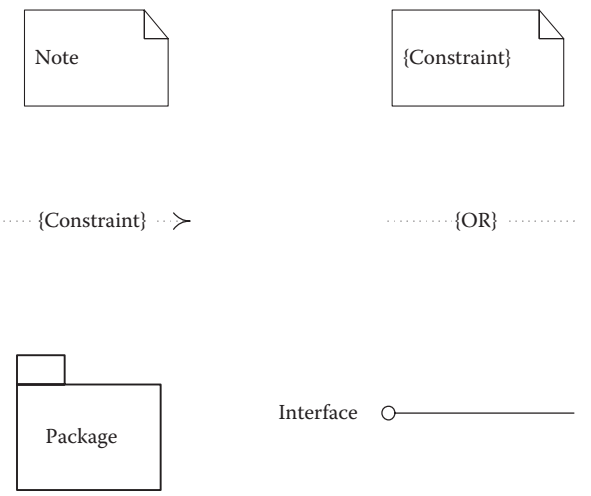


FIGURE 2.4
Additional graphical symbols available for rendering use case diagrams.

All the constraints, including general, two-element, and OR constraints, have the same categories of properties: general properties and tagged values. The general properties include the constraint name, full path, stereotype (precondition, postcondition, or invariant), language type (OCL, code, pseudocode, or text), and constraint body.

Four categories of package properties exist, including general properties, table of events, table of constraints, and tagged values. The general properties are the package name, full path, stereotype (facade, framework, stub, or system), visibility (private, protected, or public), and the indicators for *Root*, *Leaf*, and *Abstract* types of packages. The table of events contains an entry for each event. The attributes of individual events are the event name and event type (call event, signal event, change event, or time event). The table of constraints has the same format as the table of constraints for the actors, and the use cases and tagged values are just the notes for the documentation.

The interface has four categories of properties, actually a subset of the actor properties. These are general properties, table of operations, table of constraints, and tagged values. All of them are the same as the corresponding actor properties.

The requirements engineer renders the use case diagram along as they talk to the customer about the desired behavior of the system to be developed. The use case diagram is intended as a medium to communicate the requirements between the customer and the system provider. Drawing use case diagrams is simple: the right graphical symbol is selected, dragged-and-dropped to the working sheet, the corresponding properties are filled in, and they are connected to the other symbols in the sheet.

As an illustration of a use case diagram, consider a simple program for sending and receiving electronic mail messages over the Internet. The use case diagram for such a program might look like the one shown in Figure 2.5. A single actor is found in this diagram, who is the user of the program (named *User*). On the highest level of abstraction, this program has two main use cases, *Send e-mail* and *Receive e-mail*.

Both of these highest-level use cases make use of the use cases *Use DNS* (Domain Name System) and *Use TCP* (Transmission Control Protocol). The DNS service provides the mapping of the e-mail server domain name into its IP (Internet Protocol) address. The TCP provides reliable data delivery service. Other than that, the use case *Send e-mail* uses the use case *Use SMTP* (Simple Mail Transfer Protocol) and the use case *Receive e-mail* uses the use case *Use POP3* (Post Office Protocol, Version 3). Normally, an e-mail client uses SMTP to send an e-mail message to the e-mail server. Similarly, a user uses POP3 to read the e-mail messages from their mailbox.

The use case *Use DNS* uses the use case *Use IP* to send a DNS request to the DNS server and to receive DNS responses from it. The use case *Use TCP* uses the use case *Use IP* to send and receive segments of data and control

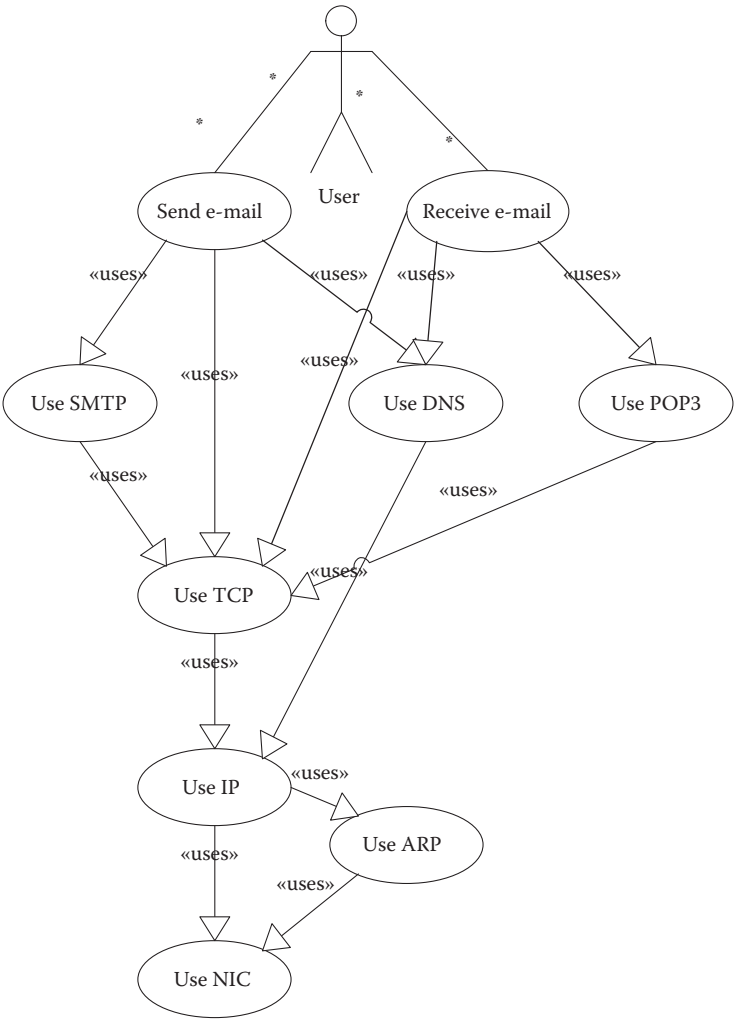


FIGURE 2.5
Use case diagram of the simple program for sending and receiving e-mails.

information over the Internet. The use case *Use IP* uses the use case *Use ARP* (Address Resolution Protocol) to map the IP address of the destination host to its physical (e.g., Ethernet) address. Alternatively, the use case *Use IP* uses the use case *Use NIC* (Network Interface Controller) to send and receive IP datagrams over the Internet. Finally, the use case *Use ARP* uses the use case *Use NIC* to send an ARP request to the ARP server and to receive an ARP response from it.

This hierarchy of use cases actually follows the hierarchy of protocols in the TCP/IP protocol stack. As already mentioned, the concept of layered software architecture, which is traditionally explained by the ISO OSI, was actually invented to enable the separation of functions and the corresponding functional requests, which are referred to as *use cases* in UML.

After creating the skeleton of the use case model, the requirements engineer must fill in the descriptions of the individual use cases. The descriptions in this example are simplified for the sake of clarity. The description of the use case *Send e-mail* in plain text is the following:

Precondition:

```
The user has issued the send mail command.
```

Main flow of events:

```
Extract the recipient's e-mail address from the e-mail message header (defined
by the RFC 822).
Extract the e-mail server domain name from the recipient's e-mail address
(string after the character "@").
Use the use case Use DNS to map the server domain name into its IP address.
Use the use case Use TCP to open the TCP connection.
Use the use case Use SMTP to send the e-mail message to the e-mail server.
Use the use case Use TCP to close the TCP connection.
Prompt the user for the next command.
```

Exceptional flow of events:

```
The user may cancel the use case at any time by issuing the cancel command.
```

Exceptional flow of events:

```
If the use case Use SMTP indicates the problem in the mail delivery, this use
case should report it to the actor User.
```

The use case *Receive e-mail* is identical to the use case *Send e-mail* with the difference being that the former uses the use case Use POP3 instead of the use case *Use SMTP*. The following description of the use case *Use DNS* is rather simple (actually, this is the description of the behavior of the DNS client):

Main flow of events:

```
Send the recursive DNS request by using Use IP.
Receive the DNS response by using Use IP.
```

The use case *Use TCP* is the active (initiator's) side of the TCP. It is defined as follows:

Main flow of events:

```
The procedure to open the TCP connection:
  Send SYN data segment.
  Receive SYN + ACK data segment.
  Send ACK data segment.
  Indicate that the connection is established.
The data transmission procedure:
  Send and receive the data segments using the sliding window.
The procedure to close the TCP connection:
  Send FIN data segment.
  Receive ACK data segment.
  Receive FIN + ACK data segment.
  Send ACK data segment.
  Indicate that the connection is closed both ways.
```

Exceptional flow of events:

```
The use case Send e-mail may close the TCP connection at any time.
```

The use case *Use SMTP* is actually the client side of the SMTP (defined by IETF RFC 821 and RFC 788) and can be described as follows (for simplicity, only one exceptional flow of events is given):

Main flow of events:

```
Receive the message 220 READY FOR MAIL.
Send the message HELLO.
Receive the message 250 OK.
Send the message MAIL FROM: <recipient's e-mail address>.
Receive the message 250 OK.
Send the message RCPT TO: <sender's e-mail address>.
Receive the message 250 OK.
Send the message DATA.
Receive the message 354 START MAIL INPUT.
Send the body of the e-mail message terminated with <CR><LF>.<CR><LF>.
Receive the message 250 OK.
Send the message QUIT.
Receive the message 221.
```

Exceptional flow of events:

```
If a use case receives the message 550 NO SUCH USER HERE, as a reply to its RCPT
TO: message, it indicates the problem to the use case Send e-mail.
```

The use case *Use POP3* is the client side of the POP3 protocol, similar to the use case *Use SMTP*. The use case *Use IP* is actually the IP protocol, which is described as follows:

Main flow of events:

```
The procedure that is used to receive the datagrams:
  Receive a datagram by using the Use NIC.
  Send the received datagram to the use case Use TCP.
The procedure that is used to send the datagrams:
  Decrement the contents of the time-to-live field of the IP datagram.
```

```
  Extract the destination IP address from the datagram header.
  Extract the destination network id from the destination IP address.
  If the destination network is local the network:
    Use the use case Use ARP to determine the physical address.
    Deliver the datagram by using the Use NIC.
  Else, route the datagram.
```

Exceptional flow of events:

```
If the datagram has been corrupted during the transmission, drop it.
```

Exceptional flow of events:

```
If the time-to-live field of the datagram counts down to 0, drop it.
```

The use case *Use ARP* is an ARP client and the use case *Use NIC* is a network card driver. The former is defined as follows:

Main flow of events:

```
Send an ARP request by using the use case Use NIC.
Receive the ARP response by using the use case Use NIC.
```

The example above, especially the use cases *Use TCP* and *Use SMTP*, should help the reader understand that a use case is a set of event sequences, not just a single sequence. To keep use cases simple, separating the main and the alternative flows of events is always desirable. Usually, we start by just writing the main flow of events for each use case and refine them later by adding the exceptional flow of events.

After this example, it should be clear that a use case captures the intended behavior of the part of the system (subsystem, class, or interface). Of course, after specifying the intended behavior, we must create a set of classes that work together to implement that behavior. The means of modeling both static and dynamic structures of the society of objects in UML are the collaboration diagrams.

2.2 Collaboration Diagrams

As already mentioned, we think of use cases as collaborations between objects. Actually, in UML we realize a use case as a collaboration of a set of objects. This concept can be explicitly shown in UML by connecting the use case with the corresponding collaboration using the realization relationship.

A **collaboration diagram** is a special kind of graph consisting of a set of vertices interconnected by a set of arcs. Basically, the vertices are the

objects and the arcs are the links that carry the messages between the interconnected objects. Additional vertices and arcs are the notes and the constraints (general constraints, two-element constraints, and OR constraints).

Collaboration diagrams are normally rendered using the appropriate graphical tools, e.g., Microsoft® Visio. This tool provides the set of graphical symbols that are placed on the working sheet by the drag-and-drop paradigm. The basic set of graphical symbols is shown in Figure 2.6. The engineer that renders the diagram must specify the properties for each instance of a symbol in the drawing.

Three categories of object properties exist: general properties, table of constraints, and tagged values. The general properties include the object name, full path, classifier name, and multiplicity. The table of constraints and the tagged values contain the same properties as the corresponding categories for the use cases (see the previous section of this chapter).

While adding objects to the collaboration diagram, we are forced to introduce the corresponding classifiers and to specify their properties (at least the classifiers' names, for a start). The classifiers have eight categories of properties, including general properties, table of attributes, table of operations, table of receptions, table of template parameters, list of the components, table of constraints, and tagged values. The general properties, the table of attributes, the table of operations, the table of constraints and tagged values contain the same properties as the corresponding categories for the use cases (see the previous section of this chapter).

The table of receptions has five columns, which contain the reception name, signal name, visibility (private, protected, or public), polymorphic indicator

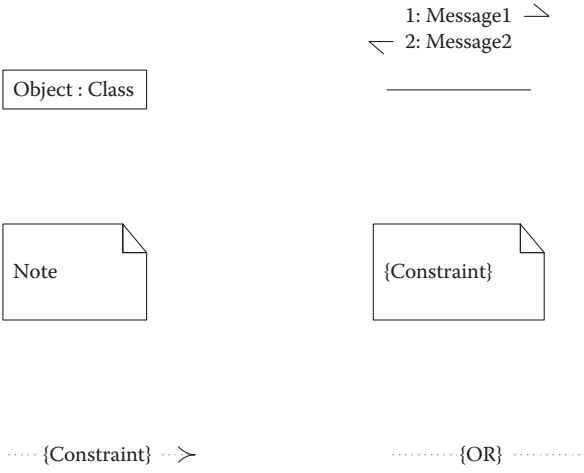


FIGURE 2.6
Set of graphical symbols available for rendering collaboration diagrams.

(false or true), and scope (classifier or instance). The table of template parameters includes the columns for the parameter name and its type. The list of components is just a list of components that implement this class.

The links in collaboration diagrams have four categories of properties, including general properties, table of messages, table of constraints, and tagged values. The general properties are the link name, its full path, and the table of link ends roles, which has two columns, the end name and its stereotype (none, association, global, local, parameter, self). The table of link messages has four columns, including the message name, its direction (forward or backward), flow kind (procedure call, flat, or asynchronous), and sequence expression. The table of constraints contains the same properties as the corresponding category of object (and classifier) properties. The tagged values are just the notes for the documentation. The notes and the constraints have the same properties as in the use case diagrams (see the previous section of this chapter).

Most frequently, we model sequential flow of control with collaboration diagrams. In this case, a message sequence expression takes the simple form of a message sequence number. However, collaboration diagrams allow modeling of more complex flows, such as iteration and branching. Iteration is modeled by prefixing the message sequence number with the iteration expression

*[<*control variable*> := <*start value*>..<*end value*>]

e.g., *[*j* := 1..*m*].

Branching is modeled by prefixing the message sequence number with the condition clause [<*condition*>], e.g., [*i* > 10]. Alternate paths of the branch have the same message sequence number prefixed by the unique non-overlapping condition, where the set of conditions must cover all the possibilities.

Next, we illustrate the use of collaboration diagrams in the example of a simple program for sending and receiving electronic mail messages over the Internet, which was introduced and modeled in the previous section of this chapter. The use case diagram for this program is shown in Figure 2.5. We start by making the real collaboration between objects that is a realization of the use case model, and continue with the study of virtual collaborations, which correspond to the peer-to-peer protocols present in this example.

To start, imagine that we are provided with the classifier FSM for modeling finite state machines. Clearly a single object of this class could be a realization of a single use case, as shown in Figure 2.5. The assumption that each use case is materialized by a single FSM object leads to a real collaboration between objects, shown in Figure 2.7.

In this class diagram, the object *mailc* (abbreviation for a mail client) is the <<*boundary class*>> object. All other objects are the <<*control class*>>

objects. The e-mail message itself would be the *<<entity object>>*, but it is not shown in Figure 2.7. Obviously, the realization of the individual use cases is as follows:

- The object *sender* is a realization of the use case *Send e-mail*.
- The object *receiver* is a realization of the use case *Receive e-mail*.
- The object *dnsc* (abbreviation for a DNS Client) is a realization of the use case *Use DNS*.
- The object *tcpc* (abbreviation for a TCP Client, i.e., the side that initiates the establishment of the TCP connection) is a realization of the use case *Use TCP*.

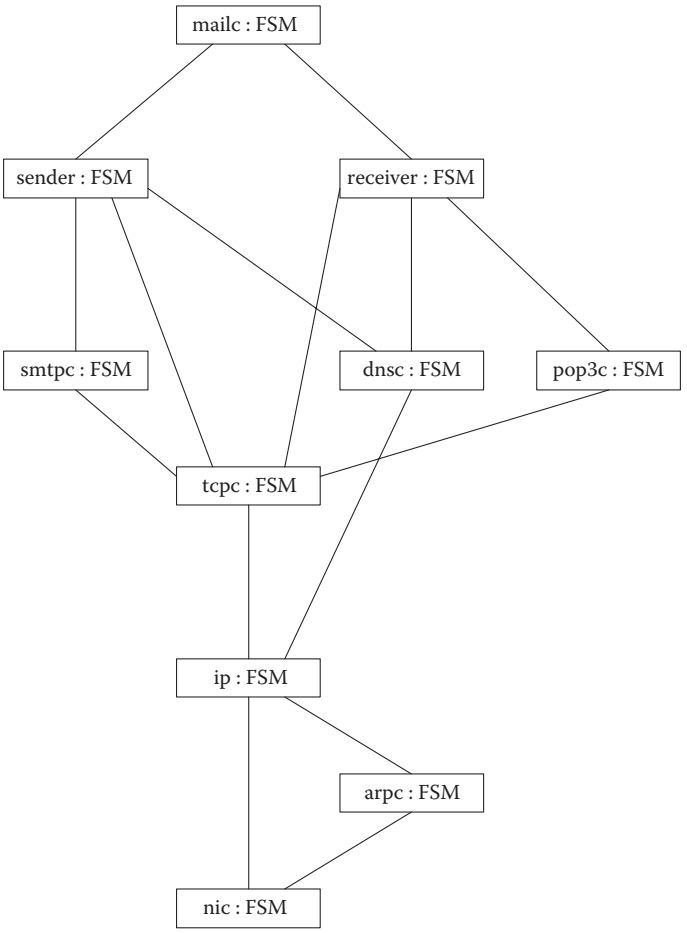


FIGURE 2.7
Collaboration diagram of the simple program for sending and receiving e-mails.

- The object *smtpc* (abbreviation for an SMTP Client) is a realization of the use case *Use SMTP*.
- The object *pop3c* (abbreviation for a POP3 Client) is a realization of the use case *Use POP3*.
- The object *ip* is a realization of the use case *Use IP*.
- The object *arpc* (abbreviation for an ARP Client) is a realization of the use case *Use ARP*.
- The object *nic* is a realization of the use case *Use NIC*.

Figure 2.7 shows general collaboration among the relevant objects, i.e., it just shows the links between objects. Essentially, it shows the software architecture. We may think of it as a family of particular collaborations. For example, the user of the program might select the use case *Send e-mail* and this would lead to a particular collaboration, or the user might select the use case *Receive e-mail* and that would lead to another particular collaboration.

Another important thing to notice and remember is that Figure 2.7 shows only the objects of the system under development. In this case, it is a program that runs on a computer connected to the Internet over its network interface card. If we want the overall picture, we can also add the models of the systems with which our system under development would normally communicate. By adding the models of these external systems, we are modeling end-to-end collaborations.

The system under development communicates with external servers, including the ARP server, the DNS server, and the e-mail server. If we assume that all of these servers run on the same computer, the model of the external environment of the system under development is rather simple (Figure 2.8). The external objects are as follows:

- The object *smtps* is the SMTP server.
- The object *pop3s* is the POP3 server.
- The object *tcps* is the TCP server, i.e., the side that accepts the establishment of the TCP connection.
- The object *dnss* is the DNS server.
- The object *arps* is the ARP server.
- The object *ips* is an instance of IP.
- The object *nics* is an instance of NIC.

The overall collaboration that corresponds to the main flow of events of the use case *Send e-mail*, up to the point when the SMTP client receives the message 220 READY FOR MAIL, is shown in Figure 2.9. The flow of events is as follows:

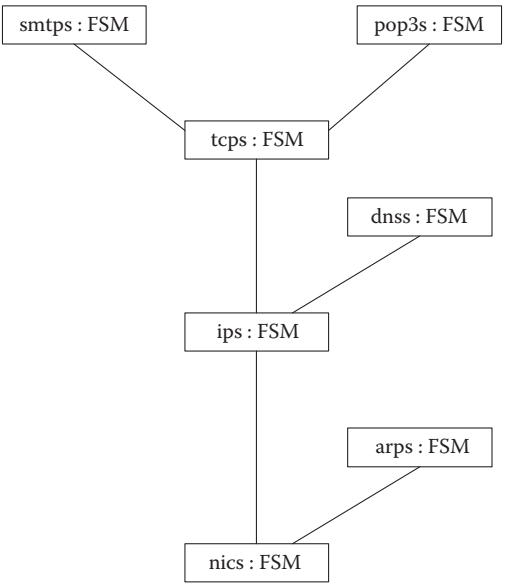


FIGURE 2.8
Collaboration diagram of the e-mail and DNS server.

1: The object *mailc* sends the signal *sendMail(msg)* to the object *sender*. The signal parameter *msg* is the e-mail message itself.

2: The object *sender* sends the signal *domainToIP(domain)* to the object *dnsc*. The signal parameter *domain* is the domain name of the e-mail server.

3: The object *dnsc* sends the signal *dnsReq(domain)* to the object *ip*. The signal *dnsReq* is actually the DNS service request message.

4: The object *ip* sends the signal *data(dnsReq)* to the object *nic*. The general signal *data* is an IP datagram. Together with the parameter *dnsReq*, it represents the datagram carrying the DNS service request message.

5: The object *nic* sends the signal *frame(dnsReq)* to the object *nics*. The general signal *frame* is a data frame from the underlying physical network (e.g., Ethernet). The signal *frame(dnsReq)* is the data frame carrying the datagram that encapsulates the DNS service request message.

6: The object *nics* sends the signal *data(dnsReq)* to the object *ips*.

7: The object *ips* sends the signal *dnsReq(domain)* to the object *dnss*.

8: The object *dnss* sends the signal *dnsRsp(ip)* to the object *ips*. The signal *dnsRsp* is the DNS service response message and its parameter *ip* is the IP address of the target e-mail server.

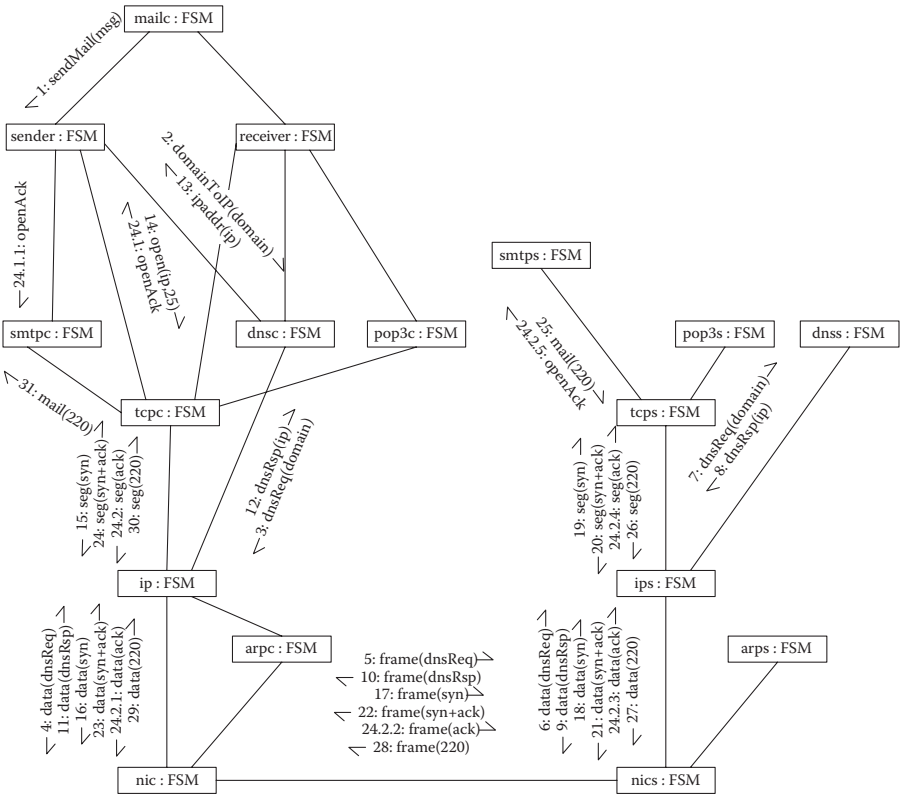


FIGURE 2.9
Overall real collaboration of the simple program for sending and receiving e-mails and its environment.

9: The object *ips* sends the signal *data(dnsRsp)* to the object *nics*.

10: The object *nics* sends the signal *frame(dnsRsp)* to the object *nic*.

11: The object *nic* sends the signal *data(dnsRsp)* to the object *ip*.

12: The object *ip* sends the signal *dnsRsp(ip)* to the object *dnsc*.

13: The object *dnsc* sends the signal *ipaddr(ip)* to the object *sender*.

14: The object *sender* sends the signal *open(ip,25)* to the object *tcpc*. The signal *open* is an active open request to TCP (TCP should send the SYN segment to initiate the TCP connection establishment procedure). Its parameters, *ip* and *25*, are the IP addresses of the target email sever and the well-known TCP port number reserved for the SMTP, respectively.

15: The object *tcpc* sends the signal *seg(syn)* to the object *ip*. The general signal *seg* is a TCP segment. The signal *seg(syn)* is a SYN (synchronization) TCP segment (i.e., it has the SYN bit set in the code field).

16: The object *ip* sends the signal *data(syn)* to the object *nic*.

17: The object *nic* sends the signal *frame(syn)* to the object *nics*.

18: The object *nics* sends the signal *data(syn)* to the object *ips*.

19: The object *ips* sends the signal *seg(syn)* to the object *tcps*.

20: The object *tcps* sends the signal *seg(syn+ack)* to the object *ips*. The signal *seg(syn+ack)* is a SYN+ACK (synchronization and acknowledgment) TCP segment (i.e., it has both SYN and ACK bits set in the code field).

21: The object *ips* sends the signal *data(syn+ack)* to the object *nics*. The signal *data(syn+ack)* is the IP datagram that encapsulates the SYN+ACK TCP segment.

22: The object *nics* sends the signal *frame(syn+ack)* to the object *nic*. The signal *frame(syn+ack)* is the data frame carrying the IP datagram that encapsulates the SYN+ACK TCP segment.

23: The object *nic* sends the signal *data(syn+ack)* to the object *ip*.

24: The object *ip* sends the signal *seg(syn+ack)* to the object *tcpc*. (The event flow now forks into two parallel flows.)

- 24.1: The object tcpc sends the signal openAck to the object sender. (The first flow begins here.)
  - 24.1.1: The object *sender* sends the signal *openAck* to the object *smtpc* (The first flow ends here.)
- 24.2: The object *tcpc* sends the signal *seg(ack)* to the object *ip*. (The second flow begins here.)
  - 24.2.1: The object *ip* sends the signal *data(ack)* to the object *nic*.
  - 24.2.2: The object *nic* sends the signal *frame(ack)* to the object *nics*.
  - 24.2.3: The object *nics* sends the signal *data(ack)* to the object *ips*.
  - 24.2.4: The object *ips* sends the signal *seg(ack)* to the object *tcps*.
  - 24.2.5: The object *tcps* sends the signal *openAck* to the object *smtps*.

25: The object *smtps* sends the signal *mail(220)* to the object tcps. The general signal *mail* is the SMTP message. The particular signal *mail(220)* is actually the message 220 READY FOR MAIL, where the first three digits are mandatory and the rest of the message is a human-readable comment. (Note: We have restarted the message numbering here for brevity.)

26: The object *tcps* sends the signal *seg(220)* to the object *ips*.

27: The object *ips* sends the signal *data(220)* to the object *nics*.

28: The object *nics* sends the signal *frame(220)* to the object *nic*.

29: The object *nic* sends the signal *data(220)* to the object *ip*.

30: The object *ip* sends the signal *seg(220)* to the object *tcpc*.

31: The object *tcpc* sends the signal *mail(220)* to the object *smtpc*. (The example ends here.)

What we have just described is the real collaboration between objects within the system under development as well as with the relevant objects in its surroundings. The real collaboration for any nontrivial system could be rather complex. This behavior should be clear from the previous example, where we intentionally stopped at the certain point of the event flow, which was selected as a compromise between showing enough complexity and maintaining clarity.

The complete list of events for the use case *Send e-mail* is much longer than the one given above. For modeling the transfer of the rest of the SMTP messages (12 of them), we would need additional 84 (12 × 7) UML events, almost three times more than already in the list above. This complexity is why we try to break the system down into its parts and analyze them in detail later.

One important aspect of the simplification is the definition of the Application Programming Interfaces (API). For example, we may define the API between the *sender* and the hierarchically lower level objects (*dnsc*, *smtpc*, and *tcpc*), or the API between *tcpc* and *ip*, and so on. Other important items are the virtual collaborations that are governed by the peer-to-peer protocols. Consider for example the virtual collaboration between *dnsc* and *dnss* (Figure 2.10). The corresponding flow comprises only two events, *dnsReq(domain)* and *dnsRsp(ip)*.

The virtual collaboration between *tcpc* and *tcps* is governed by the TCP. It is slightly more complex and comprises the following flow of events (Figure 2.11):

1: The object *tcpc* sends the signal *seg(syn)* to the object *tcps*.

2: The object *tcps* sends the signal *seg(syn+ack)* to the object *tcpc*.

3: The object *tcpc* sends the signal *seg(ack)* to the object *tcps*.

4: The object *tcpc* sends the signal *seg(data)* to the object *tcps*. (Data transmission phase)

5: The object *tcpc* sends the signal *seg(fin)* to the object *tcps*.

6: The object *tcps* sends the signal *seg(ack)* to the object *tcpc*.

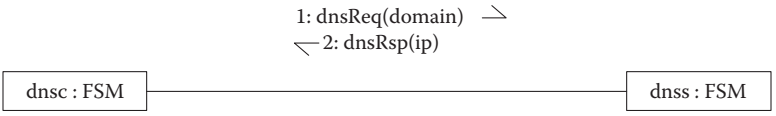


FIGURE 2.10
Virtual collaboration between the DNS client and the DNS server.

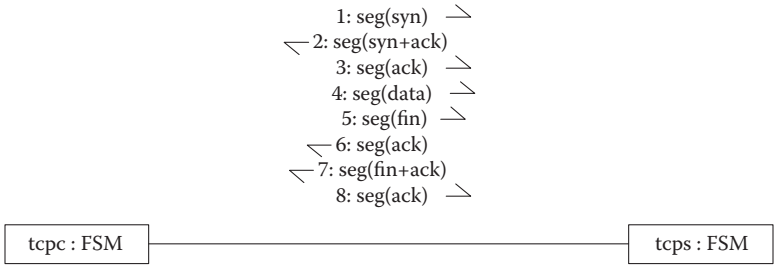


FIGURE 2.11
Virtual collaboration between two TCP entities.

7: The object *tcps* sends the signal *seg(fin+ack)* to the object *tcpc*.
8: The object *tcpc* sends the signal *seg(ack)* to the object *tcps*.

Finally, the virtual collaboration between *smtpc* and *smtps* (in accordance with SMTP) is of the same order of complexity (Figure 2.12; note that only the first eight events are shown in the figure). The corresponding flow of events is the following:

1: The object *smtps* sends the signal *mail(220)* to the object *smtpc*.
2: The object *smtpc* sends the signal *mail(HELO)* to the object *smtps*.
3: The object *smtps* sends the signal *mail(250_OK)* to the object *smtpc*.
4: The object *smtpc* sends the signal *mail(MAIL_FROM:)* to the object *smtps*.
5: The object *smtps* sends the signal *mail(250_OK)* to the object *smtpc*.
6: The object *smtpc* sends the signal *mail(RCPT_TO:)* to the object *smtps*.
7: The object *smtps* sends the signal *mail(250_OK)* to the object *smtpc*.
8: The object *smtpc* sends the signal *mail(DATA)* to the object *smtps*.
9: The object *smtps* sends the signal *mail(354_START_MAIL_INPUT)* to the object *smtpc*.

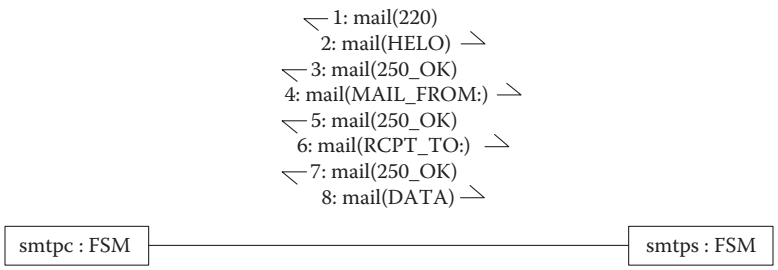


FIGURE 2.12
Virtual collaboration between the SMTP client and the SMTP server.

10: The object *smtpc* sends the signal *mail(MAIL_BODY)* to the object *smtps*.
11: The object *smtps* sends the signal *mail(250_OK)* to the object *smtpc*.
12: The object *smtpc* sends the signal *mail(QUIT)* to the object *smtps*.
13: The object *smtps* sends the signal *mail(221)* to the object *smtpc*.

2.3 Requirements and Analysis Example

This section of this chapter illustrates the requirements and analysis phases of communication protocol engineering with the example of a simple SIP softphone. Normally, the requirements phase starts by acquiring the relevant domain-specific knowledge and continues by the construction of the corresponding requirements model, which is the input for the analysis phase. As already mentioned, the output of the analysis phase is the corresponding analysis model. Sections 2.3.1 through 2.3.3 cover a short overview of the domain-specific information, the requirements, and the analysis models of a simple SIP softphone.

2.3.1 SIP Domain Specifics

SIP is the application layer protocol used for creating, modifying, and terminating sessions, such as Internet telephone calls and multimedia distribution and conferences, with one or more participants. It has been standardized by the IETF RFC 3261 (Rosenberg et al., 2002) and related series of RFCs (RFC 3262, RFC 3263, RFC 3264, RFC 3265, RFC 3372, RFC 3428, RFC 3485, RFC 3487, and others). In contrast to the ITU-T H.323 family of protocols—which provide the whole protocol stack for multimedia communications—SIP is just the control and signaling component on the top of the multimedia architecture.

Aside from SIP, the multimedia architecture will typically include RTP (Real-Time Transfer Protocol, RFC 1889), RTSP (Real-Time Streaming Protocol, RFC 2326), MEGACO (Media Gateway Control Protocol, RFC 3015), and SDP (Session Description Protocol, RFC 2327). SIP does not provide any service on its own. Instead of full services, it provides primitives for the services that are implemented in the overall architecture. These primitives are based on an HTTP-like (Hyper Text Transport Protocol) request and response transaction model.

The main SIP abstractions are the session, the dialog, and the transaction. A multimedia session is a set of multimedia senders and receivers, as well as data streams flowing from senders to receivers. A dialog is a peer-to-peer relationship between two user agents (end points in the communication) that

persists for some time. A transaction is the collaboration between the client and the server, which comprises all the messages from the first request sent from the client to the server up to the final response sent from the server to the client. The requests are processed automatically, meaning that either all requested actions are conducted, if the request has been accepted, or none of the actions are conducted, if the request has not been accepted.

Two main transaction types exist, referred to as invite (officially written in capital letters, i.e., INVITE) and non-invite (or, more formally, non-INVITE) transactions. An invite transaction is a three-way handshake comprising the request, the response, and the acknowledgment. In contrast, a non-invite transaction is the two-way handshake starting with the request and ending with the corresponding response.

Notice that the roles of the user agents (communication end points) are not fixed, and they change on the transaction by transaction bases. The user agent that creates a new request becomes a user agent client (UAC), whereas the user agent that receives the request becomes the user agent server (UAS). Another important detail is that a new transaction (either invite or non-invite) may not be started while an invite transaction is in progress. Alternatively, a new invite transaction may be started while a non-invite transaction is in progress.

Besides user agents, the SIP standard defines three types of SIP servers, namely, the proxy server (stateful or stateless), the registrar, and the redirect server. A proxy server is the mediator that helps end points set up the session. Officially, it is an intermediary entity that acts as both a server and a client for the purpose of making requests on behalf of other clients. A registrar is a server that supports the registration of the user agents by maintaining the corresponding database for the domain it handles. This database is referred to as a location service. These two types of servers are most frequently collocated in the same physical machine. A redirect server can be viewed as a proxy server with limited capabilities. It is only capable of directing the client to contact an alternate set of Uniform Resource Identifications (URI).

Requests and responses between a server and a client are sent as SIP messages. The SIP message comprises the start line, one or more header fields, empty lines (carriage-return line-feed sequences, CRLF), and an optional message body. The start line is different in requests and in responses. In the former case, it is referred to as a request line, and in the latter as a status line. The request line comprises the method name (according to the RFC 3261, six methods are available in SIP: REGISTER, INVITE, ACK, CANCEL, BYE, and OPTIONS), the request URI, and the SIP version (currently "SIP/2.0"). The status line comprises the SIP version, the status code (a three-digit integer result code), and the reason phrase (textual status description).

The SIP protocol stack comprises four layers. Starting from the top and going down the hierarchy, these are the transaction user (TU) layer, the transaction layer, the transport layer, and the syntax and encoding of SIP messages. A transaction user is any SIP entity (client or server) except for the

stateless proxy. The transaction layer supports transactions, which are the key component of SIP. The transport layer provides for the transfer of SIP messages across the Internet. SIP may use three types of transport services, including unreliable (UDP), reliable (TCP), and encrypted (Transport Layer Security, TLS) transport service. Most of the SIP message and header field syntax is identical to HTTP/1.1. Although SIP is close to the HTTP philosophy, it is not an extension of HTTP.

As mentioned above, the SIP standard specifies six methods, including REGISTER for registering contact information, INVITE, ACK, CANCEL for setting up sessions, BYE for terminating sessions, and OPTIONS for querying servers regarding their capabilities. Any INVITE after the initial invite to the same destination is called re-INVITE and is used for modifying the session and dialog parameters. The method INVITE starts the invite transaction; all other methods start non-invite transactions. Interestingly enough, six status code types are also found, depending on the value of status code first digit, as follows:

1xx: Provisional (the request has been received and its processing has been started)

2xx: Success (the request has been successfully processed)

3xx: Redirection (further action by the client is needed)

4xx: Client error (the request contains an error or it may not have been fulfilled on this server)

5xx: Server error (the request is valid, but the server failed to fulfill it)

6xx: Global failure (the request cannot be fulfilled on any server)

As an example, consider the typical scenario of the SIP session setup in Figure 2.13. (Note: This figure is actually a UML sequence diagram. Sequence diagrams are intentionally introduced later in Chapter 3. For the moment, it is enough to assume that the rectangular symbols are the communicating entities and that the arrows are the messages they exchange. Time advances downwards.) Two user agents *ua1* and *ua2*, together with their corresponding proxy servers *p1* and *p2*, constitute the SIP trapezoid (imagine the trapezoid by "drawing" the lines that connect *ua1*, *p1*, *p2*, and *ua2*).

Suppose that *ua1* wants to set up a session with *ua2*. It starts by sending an invite request to the proxy server that is responsible for its domain, and that is *p1*. Proxy *p1* locates the proxy server responsible for the destination *ua2*, namely *p2*, and forwards the invite request to it. At the same time, *p1* sends back the response 100 TRYING to *ua1*. Proxy *p2* locates the destination user agent, *ua2*, forwards the invite request to it, and sends back the response 100 TRYING to the proxy *p1*. *ua2* receives the invite request and sends back the response 180 RINGING, which is forwarded by the proxies *p2* and *p1* to *ua1*.

At this point, *ua2* indicates the incoming invite request to its user. The user accepts the request and *ua2* sends back the response 200 OK, which is

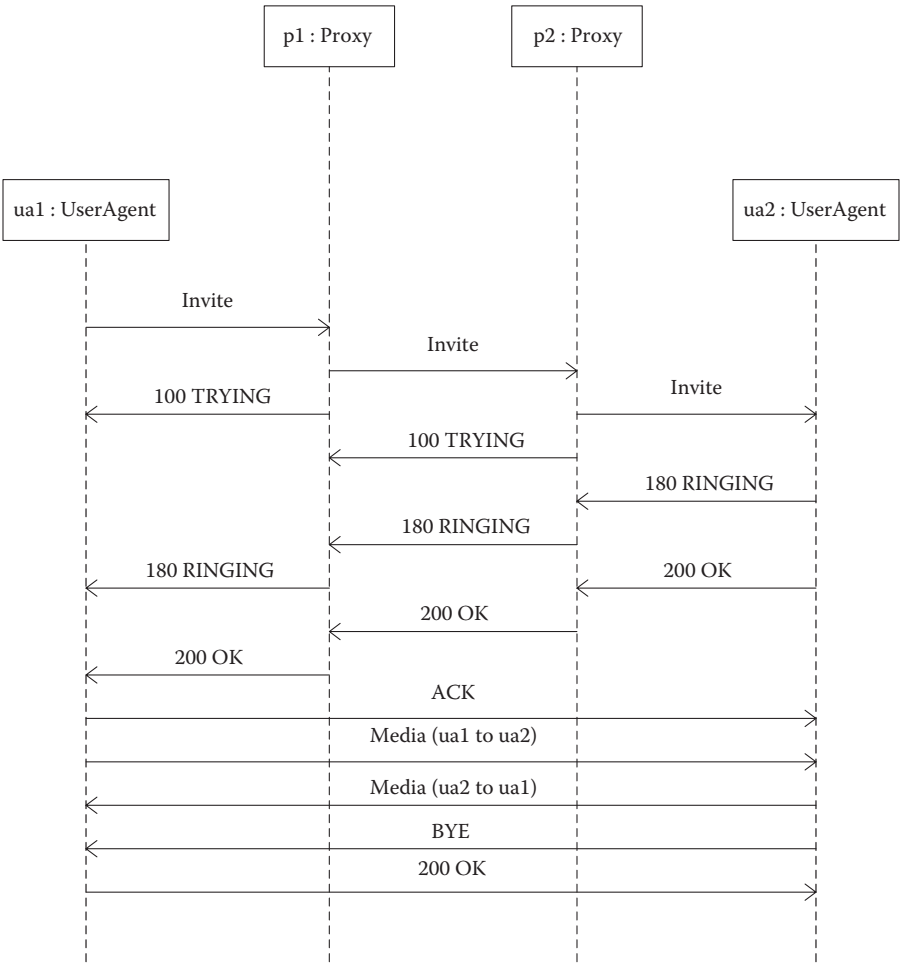


FIGURE 2.13
Example of SIP session setup (with SIP trapezoid).

forwarded by the proxies *p2* and *p1* to *ua1*. The dialog between *ua1* and *ua2* is successfully established. Further on, *ua1* sends the ACK request to *ua2* directly (the end of the three-way handshake). The session is successfully established at this point. The communicating user agents may now exchange the media. In reality, the media is exchanged in the full-duplex mode, i.e., both sides may send data to the other side simultaneously. Unfortunately, in UML sequence diagrams we cannot model the full-duplex communication, because only unidirectional messages may be used. Therefore, we represent the media exchange by the two separate messages, namely by the message Media (*ua1* to *ua2*) and the message Media (*ua2* to *ua1*).

The session may be terminated by either *ua1* or *ua2*. Suppose that *ua2* wants to terminate the session. It sends the BYE request to *ua1* directly, which in its turn sends back the response 200 OK. The session is successfully closed. This is an example of the non-invite transaction.

This simplified explanation hides one rather important aspect of the invite three-way handshake, and that is the application of the offer-answer procedure. This procedure is used by *ua1* and *ua2* to determine the session parameters in accordance with SDP. The first offer must be carried either by the invite request or by the response 200 OK. If the offer is carried by the invite request (*ua1* makes the first offer), the answer must be included in the response 200 OK. If the offer is carried by the response 200 OK (*ua2* makes the first offer), the answer must be included in the ACK request (the last message in the three-way handshake). The session is successfully established only after the offer-answer procedure is successfully ended.

2.3.2 SIP Softphone Requirements Model

SIP softphone is the application that normally runs on some computer—for example, a desktop PC—and enables its user to set up multimedia sessions and to communicate with other SIP users or entities over the Internet. Such an application would typically use some type of graphical user interface (GUI) and device drivers for the sound card and the web camera, typically provided by the local operating system (out of scope for this book) and, of course, the SIP protocol stack.

This section shows how to construct the requirements model for the SIP protocol stack in a simple SIP softphone. As mentioned previously, the SIP protocol stack comprises the transaction user layer, the transaction layer, and the transport layer. In terms of use cases, the user uses the application (softphone), which in turn uses both the transaction layer and the transport layer. The transaction layer also uses the transport layer. The use case diagram shown in Figure 2.14 is a simple requirements model that captures these relations.

We can refine this simple model by taking into account the details of the individual layers of the SIP protocol stack. To start, the transaction user (TU) layer dynamically creates and uses the user agent clients (UAC) and the user agent servers (UAS) entities to support outgoing and incoming invite requests. Both UAC and UAS use the transaction layer (TAL), as well as the transport layer, which is accessible through the transport layer interface (TLI). TAL and TLI are abbreviations introduced here (they have not been taken from the RFC 3261).

Similar to TU, TAL dynamically creates and uses invite client transactions (INVITE CT), non-invite client transactions (non-INVITE CT), invite server transactions (INVITE ST), and non-invite server transactions (non-INVITE ST). TAL and all transactions use TLI, but they are all also used by

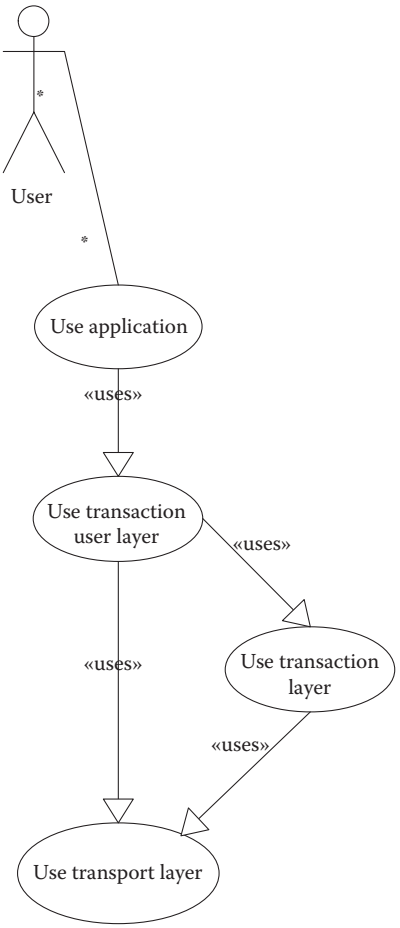


FIGURE 2.14
Use case diagram of the simple SIP softphone.

TU. Finally, TLI uses UDP, TCP, or TLS. The detailed use case diagram of the simple SIP softphone is shown in Figure 2.15.

Before proceeding further, two important points must be emphasized. The first is that the direct relations between TU and TLI are strictly in accordance with the RFC 3261, although this may seem to be an error because it violates the ISO OSI ideal of a strictly layered architecture (no direct communication between layer $i + 1$ and layer i). The second point is that the relations between TU and transactions, and transactions and TLI, are not prescribed by the RFC 3261 but they are also not forbidden. These relations are introduced to minimize the message paths at the expense of the increased relations complexity.

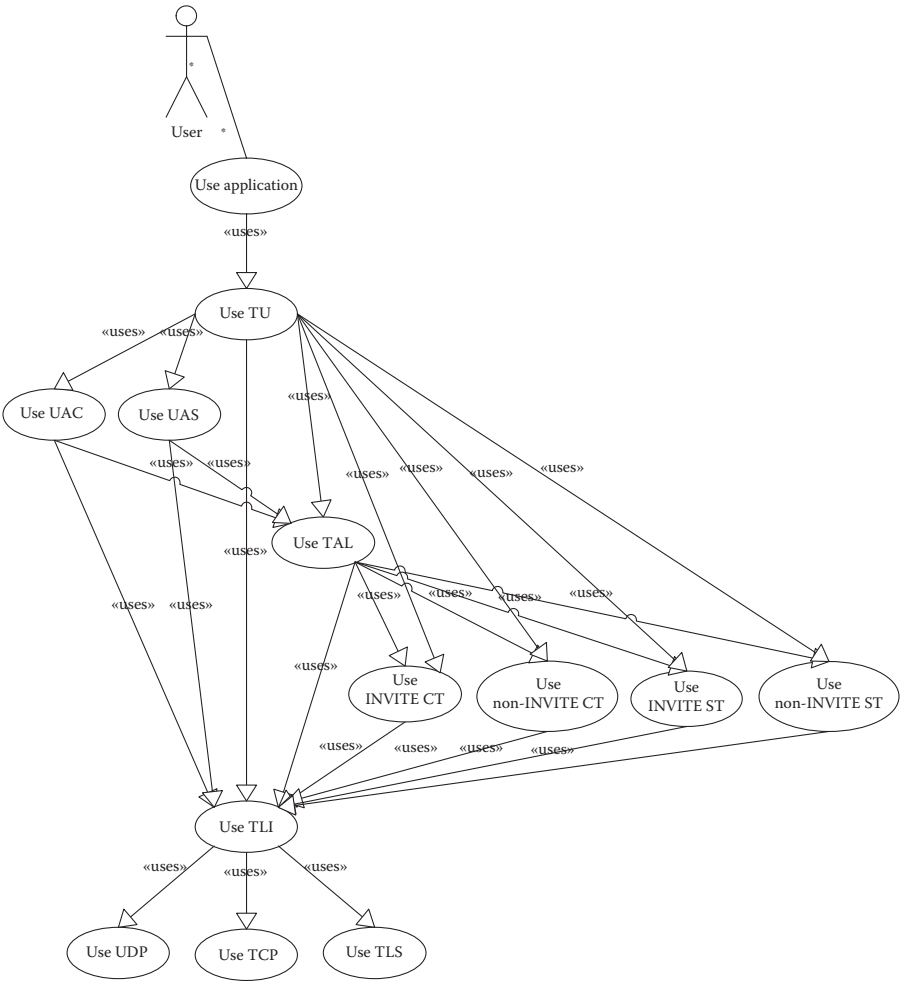


FIGURE 2.15
Detailed use case diagram of the simple SIP softphone.

To complete the requirements model, we need to describe the individual use cases. The use case *Use application* is actually the main program that interacts with the user and makes use of the SIP protocol stack and is out of the scope of this book. The use case *Use TU* is responsible for dispatching TU messages (coming from the application and the lower layers and going to the user agent clients and servers and to the application), as well as for dynamic creation of user agent clients and servers.

The use case *Use UAC* provides a set of procedures for the client side of the transactions. The high-level description of these procedures follows:

Main flow of events:

```
Receive the request from the application.
Dispatch it to the corresponding procedure.
Registration procedure:
  Create and send REGISTER request.
  Receive the response.
  Indicate the response to the application.
Session setup procedure:
  Create and send INVITE request.
  Receive provisional responses (1xx), if any.
  Receive the final response (not 1xx).
  Indicate the final response to the application.
  If the final response is 2xx,
    Send ACK request.
Cancel session setup procedure:
  If the final response has not been received,
    Create and send CANCEL request.
    Receive the response.
    Indicate the response to the application.
Modify session/dialog procedure:
  Perform session setup procedure.
Query server capabilities procedure:
  Create and send OPTIONS request.
  Receive the response.
  Indicate the response to the application.
Terminate session procedure:
  Create and send BYE request.
  Receive the response.
  Indicate the response to the application.
```

The use case above includes only the main flow of events. A more detailed version would also include the exceptional flow of events that would describe the time management and the retransmissions of the unacknowledged SIP messages. These are skipped here for brevity (in reality, we also start from a very simple version of use cases and refine them later). The same is true for all the other use cases given in this subsection.

The use case *Use UAS* provides the set of procedures for the server side of the transactions. The high-level description of these procedures is as follows (the implementation is rather simple and takes the passive, goodwill approach).

Main flow of events:

```
Receive the request from the TU dispatcher (i.e., remote SIP entity).
Dispatch it to the corresponding procedure.
Session setup service procedure:
  Receive the incoming INVITE request.
  Indicate INVITE request to the application.
  Send the provisional response, e.g., 180 RINGING.
  If the user accepts the call,
    Send the final response 200 OK.
    Receive ACK request.
Cancel session setup service procedure:
  Receive CANCEL request.
  Send the final response 200 OK.
  Report the outcome to the application.
```

```
Modify session/dialog service procedure:
  Receive INVITE request.
  Send the final response 200 OK.
  Report the outcome to the application.
Query server capabilities service procedure:
  Receive OPTIONS request.
  Send the final response 200 OK.
  Report the outcome to the application.
Terminate session service procedure:
  Receive BYE request.
  Send the final response 200 OK.
  Report the outcome to the application.
```

The use case Use TAL is responsible for dispatching TAL messages (coming from TU, UAC, UAS, and TLI and going to the TAL transactions), as well as for dynamic creation of TAL transactions. The use case Use INVITE CT is an invite client transaction. Its description is as follows:

Main flow of events:

```
Receive INVITE request from TAL.
Forward INVITE request to TLI.
Receive 1xx response from TAL.
Forward 1xx response to TU.
Receive the final response from TAL.
Forward the final response to TU.
If the final response is 3xx-6xx,
  Send ACK request to TLI.
```

The use case *Use INVITE ST* is an invite server transaction. Its description is as follows:

Main flow of events:

```
Receive INVITE request from TAL.
Forward INVITE request to TU.
Receive 1xx response from TAL.
Forward 1xx response to TLI.
Receive the final response from TAL.
Forward the final response to TLI.
```

The use case *Use non-INVITE CT* is a non-invite client transaction. Its description is as follows:

Main flow of events:

```
Receive the request from TAL.
Forward the request to TLI.
Receive the response from TAL.
Forward the response to TU.
```

The use case *Use non-INVITE ST* is a non-invite server transaction, which is defined as follows:

Main flow of events:

```
Receive the request from TAL.
Forward the request to TU.
Receive the response from TAL.
Forward the response to TLI.
```

The use case *Use TLI* is responsible for dispatching transport messages. It routes the requests from upper layers toward its remote peer in a forward direction, and routes the responses received from its remote peer toward the upper layers in a backward direction (non-ACK responses are sent to TAL, whereas ACK responses are sent to TU). It may use UDP, TCP, or TLS for the communication with its peers over the Internet. The description of this use case is as follows:

Main flow of events:

```
Receive a request from upper layers.
Send the request to the remote peer.
Receive the response from the remote peer.
If the response is ACK,
  Send it to TU,
Else,
  Send it to TAL.
```

Now that we have completed the use case diagram, we can proceed to the next engineering phase. This phase is the analysis, whose main goal is the definition of the software architecture.

2.3.3 SIP Softphone Analysis Model

Generally, the analysis model is constructed by defining the collaboration in a set of objects for each use case in the source requirements model. This process becomes obvious when considering the rough use case diagram shown in Figure 2.14. However, by refining the use cases, we may reach a point when a single class can realize a single use case. Figure 2.15 is an example of exactly such a use case diagram. Each use case is rather simple, so that a single class can realize it. Along this approach, assume the following mapping:

- The instance of the class *FSM* named *app* realizes the use case *Use application*.
- The instance of the class *TUDisp* named *tud* realizes the use case *Use TU*.
- An unnamed instance of the class *UAClient* realizes the use case *Use UAC*.
- An unnamed instance of the class *UAServer* realizes the use case *Use UAS*.

- The instance of the class *TALDisp* named *tald* realizes the use case *Use TAL*.
- An unnamed instance of the class *InClientT* realizes the use case *Use INVITE CT*.
- An unnamed instance of the class *NIClientT* realizes the use case *Use non-INVITE CT*.
- An unnamed instance of the class *InServerT* realizes the use case *Use INVITE ST*.
- An unnamed instance of the class *NIServerT* realizes the use case *Use non-INVITE ST*.
- The instance of the class *TLIDisp* named *tlid* realizes the use case *Use TLI*.
- The instance of the class *FSM* named *udp* realizes the use case *Use UDP*.
- The instance of the class *FSM* named *tcp* realizes the use case *Use TCP*.
- The instance of the class *FSM* named *tls* realizes the use case *Use TLS*.

The mapping above translates the use case diagram (shown in Figure 2.15) into the general collaboration diagram (shown in Figure 2.16). This diagram actually shows the software architecture, which defines the software objects that constitute the software system or product and the associations among them.

The software architecture can be used for the further study of particular object collaborations to check if the architecture is feasible and, if not, to refine the use case or collaboration diagram. An example of a particular collaboration is shown in Figure 2.17. This diagram shows the handling of the invite request initiated by the softphone user. The flow of events is as follows:

1: The object *app* sends the event *inviteReq(adr)* to the object *tud*.

2: The object *tud* sends the event *inviteReq(adr)* to an unnamed instance of the class *UAClient*.

3: The unnamed instance of the class *UAClient* sends the event *req(INVITE)* to the object *tald*.

4: The object *tald* sends the event *req(INVITE)* to an unnamed instance of the class *IClientT*.

5: The unnamed instance of the class *IClientT* sends the event *req(INVITE)* to the object *tlid*.

6: The object *tlid* sends the event *req(INVITE)* to its peer over the object *tcp*.

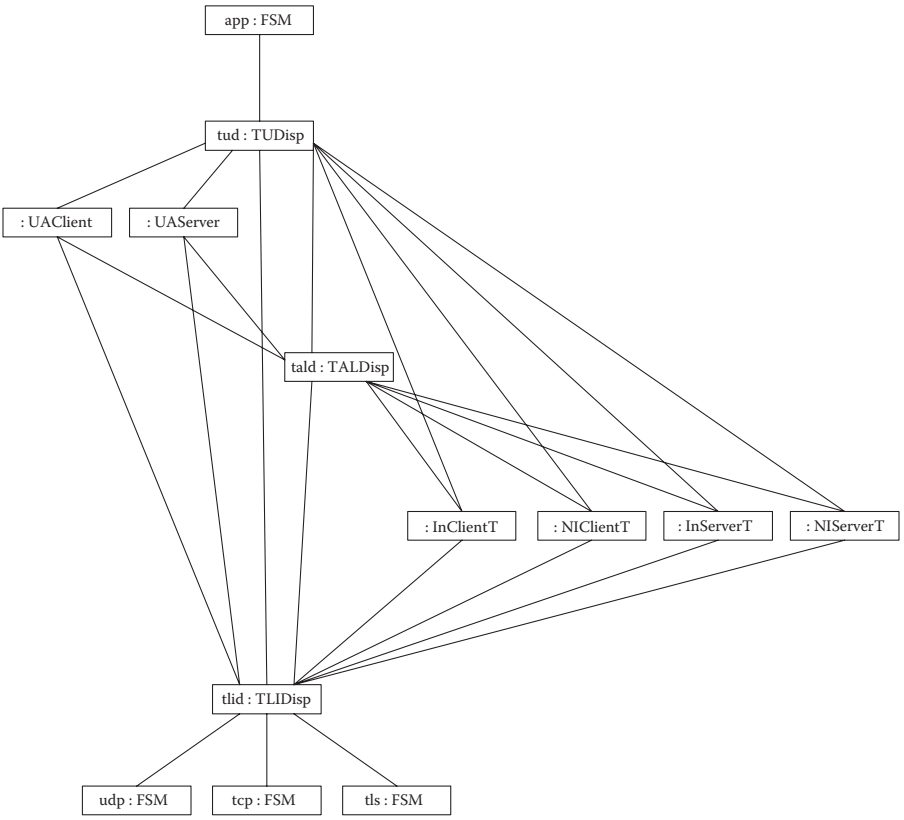


FIGURE 2.16
General collaboration diagram of the simple SIP softphone.

7: The object *tlid* receives the event *rsp(1xx)* from its peer over the object *tcp*.

8: The object *tlid* sends the event *rsp(1xx)* to the object *tald*.

9: The object *tald* sends the event *rsp(1xx)* to an unnamed instance of the class *IClientT*.

10: The unnamed instance of the class *IClientT* sends the even *rsp(1xx)* to the object *tud*.

11: The object *tud* sends the event *rsp(1xx)* to an unnamed instance of the class *UAClient*.

12: The object *tlid* receives the event *rsp(200)* from its peer over the object *tcp*.

13: The object *tlid* sends the event *rsp(200)* to the object *tald*.

14: The object *tald* sends the event *rsp(200)* to an unnamed instance of the class *IClientT*.

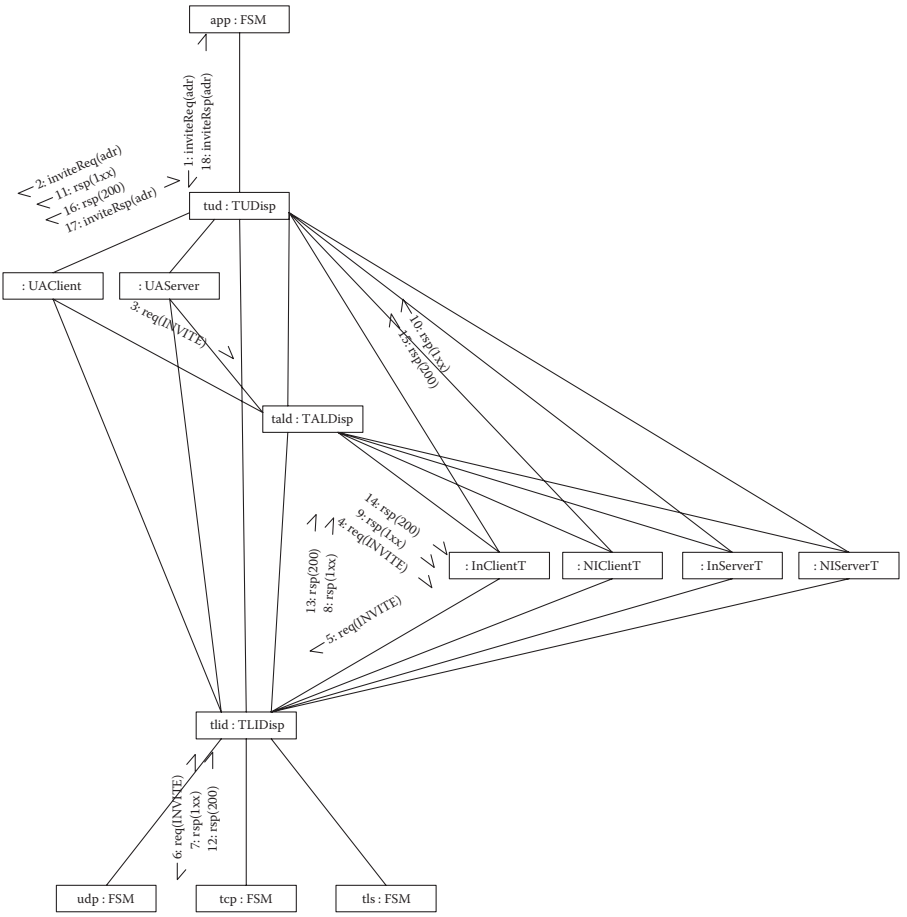


FIGURE 2.17
Collaboration diagram showing the part of the SIP session setup.

15: The unnamed instance of the class *IClientT* sends the event *rsp(200)* to the object *tud*.

16: The object *tud* sends the event *rsp(200)* to an unnamed instance of the class *UAClient*.

17: The unnamed instance of the class *UAClient* sends the event *inviteRsp(adr)* to the object *tud*.

18: The object *tud* sends the event *inviteRsp(adr)* to the object *app*.

Generally, *req()* and *rsp()* designate SIP requests and SIP responses in the flow of events shown above. For example, *req(INVITE)* is the SIP invite request, *rsp(1xx)* is the SIP provisional response, and *rsp(200)* is the SIP final response.

References

Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1998.

Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1998.

Broekman, B. and Notenboom, E., *Testing Embedded Software*, Addison-Wesley, London, 2003.

Meyer, S. and Apfelbaum, L., "Use Cases Are Not Requirements," http://www. geocities .com/model_based_testing/online_papers.htm, 1999.

Rosenberg, J. et al., "RFC 3261–SIP: Session Initiation Protocol," http://www.faqs.org /rfcs/rfc3261.html, 2002.

3

Design

System **design** is a phase in engineering work that follows the system requirements and analysis phases. Its main goal is to synthesize a complete solution based on the result of the analysis phase (obtaining the analysis model of the system), which is actually a rough architecture—a skeleton—of the system. We can imagine the system synthesis as a process of creating the body of the system. This body is a reflection of the details related to the system structure and its behavior.

Note that the complete solution of the system mentioned above is not the system itself, but rather a detailed vision of the system that comprises all the details sufficient to construct the system. Technically, we refer to this vision as a **design model**. Therefore, the system synthesis is a process that takes an analysis model as its input and produces the design model as its output.

The design model defines the two most important system aspects:

- System structure
- System behavior

The **system structure** defines the elements of the system and their associations. Sometimes it is referred to as the **static structure** because it defines the static view of the system, i.e., a view without any respect to time. The **system behavior** defines the outputs of the systems as functions of time or their inputs. In the case of a family of communication protocols, which are most frequently modeled as groups of finite state machines (automata), the static structure defines the automata and the links between them whereas the system behavior defines the state transitions for the individual automata and the external messages.

Besides system synthesis, or system design, the communication protocol design phase described in this book includes two additional designs, namely **deployment design** and **test design**, which result in a **deployment model** and a **test model**, respectively. The main goals of the deployment design are identifying network nodes and configurations as well as identifying design subsystems and interfaces. The deployment model is especially important for the complex communication systems comprising many distributed components. For less complex systems, it is not as important, and for very simple systems it may not even be necessary.

Although the system design and deployment models make the complete vision of the system, they do not specify how the system can be verified. Therefore, the engineers conduct the test design by taking the requirements and design models and creating a test model. The test model actually defines the behavior of the testers, who emulate the environment of the system. As already mentioned in the previous chapter, the test model is most frequently referred to as a test suite, which comprises a set of test cases. Each test case specifies a series of test input values (events and messages) to the system and the corresponding output values (events and messages) that are expected at the system output as the results of correct system reactions to the given series.

To summarize, a communication protocol design is a process that takes the requirements and analysis as its input and provides the following models as its output:

- System design model
- System deployment model
- System test model

The means of making these models today are UML diagrams or some domain-specific languages, which are introduced in this chapter. The design engineer starts from the analysis model, essentially a collaboration of <<*boundary*>>, <<*control*>>, and <<*entity*>> classes, described in the corresponding collaboration diagram. The development model is made by mapping each class from the analysis model to a set of new classes in the development model. If the analysis model is well refined, this might even be a one-to-one mapping or close to it. For example, the analysis model of the SIP softphone given at the end of the previous chapter is detailed enough, and the corresponding collaboration diagram is a good base for the refinements that must be made during the system design phase.

The means of defining the static structure of the system in UML are class diagrams and object diagrams. A **class diagram** shows the design classes and the static relations (dependencies, associations, and generalizations) among them without any respect to time. It shows important details about classes, such as their members, fields and functions, types, visibility, and so on. The **object diagram** is similar to the class diagram except that it shows the system frozen at a certain moment of time. Typically, the object diagram will show system objects (class instances) with the characteristic and important values of certain field members.

The means of gathering and refining details about the system behavior are the UML interaction diagrams. Two types of interaction diagrams are found, namely collaboration diagrams (introduced in the previous chapter) and **sequence diagrams**. Collaboration diagrams show the interaction

organized by the architecture, meaning that their focus is an architectural view of the system. The architecture is adorned by the flow of events. The sequence of events is shown by adding sequence numbers as prefix labels to the events.

Alternately, sequence diagrams show system interactions from a time progress perspective. The top of the sequence diagram shows the objects of the system without static relations among them. Each object is represented further by a vertical line rendered from its bottom toward the bottom of the diagram. Time advances in the same direction. The interaction itself is shown by the series of events and messages sent among the objects, which are rendered by horizontal arrows from the source object's line to the destination object's line.

The means of specifying complete system behavior are **activity diagrams** and **statechart diagrams** or, more briefly, **statecharts**. An activity diagram shows the action or activity states, starting from the initial and ending in the final state. State transitions can be sequential, branching, or concurrent (through forking and joining). The activity diagram is essentially a flowchart that emphasizes the activity that takes place over time, similar to PERT charts.

Statecharts are the means of specifying finite state machines in UML. They are a type of advanced state transition graphs. A statechart shows simple and composite states starting from the initial and ending in the final state. The composite states are a means to organize states hierarchically. The state transitions can be guarded by conditions and they can indicate firing events and the corresponding actions.

The main goal of the deployment design is the decomposition of the system in two dimensions. Horizontally, the system is partitioned into parts that are deployed onto different network nodes. The term used for nodes by ISO OSI is open systems. Vertically, the system is partitioned into layers. Typical layers recognized by the USDP are the following:

- Application-specific layer
- Application-general layer (e.g., packages common for a set of applications)
- Middleware layer (e.g., Java VM and Java packages)
- System-software layer (e.g., TCP/IP protocol stack)

Furthermore, the system-software layer is generically partitioned by ISO OSI into the following seven layers:

- Application layer
- Presentation layer
- Session layer

- Transport layer
- Network layer
- Data link layer
- Physical layer

Another way to vertically partition is in accordance with the TCP/IP Internet layers, as follows:

- Application layer
- Transport layer
- Network layer
- Network interface layer

In the context of operating systems, we can think of layers as processes. Logically, each process has its own program and the processor that executes it but, in reality, some of the processes may share the program or the processor. The processes sharing the same program are referred to as threads. The processes sharing the same processor constitute the multiprogramming set.

The layers do not exist for themselves—rather, they are typically created to service the requests issued by the upper layers. When the number of requests increases, the engineers face the scalability problem, which can be solved by deploying the same layer on more processors. If the layers are the instances of the same class, we refer to them as replicas. Alternately, on multiprocessor systems with common memory, it might be possible for these layers to share the same program.

The deployment of horizontal system partitions onto different processors or computers is used rather frequently by system designers. Examples include the client–server architecture, the multitier architecture, and others. This convenience is why most engineers think of it in the first place when deployment issues are raised. However, the deployment of a vertical system that partitions onto various processors is also possible. A typical example is the Bluetooth Host Controller Interface (HCI), which is a demarcation line between the host processor that executes the upper layers and the Bluetooth link controller (a microprocessor, a microcontroller, or a digital signal processor) that executes the lower layers.

Horizontal and vertical system partitioning are typically conducted as two interactive activities. The designer typically partitions the horizontal system by rendering the deployment diagram, which shows the network nodes, links between them, and the subsystems deployed on individual nodes. Alternately, vertical partitioning—sometimes referred to as subsystem modeling—results in a class diagram that shows just the subsystems (packages) hierarchically organized in layers, and the dependencies among

the subsystems. These two diagrams can be combined in the overall deployment diagram, which shows both the hierarchy and the deployment.

Another important design goal is identifying and providing generic design mechanisms that handle common requirements. The generic design mechanisms can be provided as design classes, collaborations, or subsystems. Examples of the generic design mechanisms in communication protocol engineering are:

- Protocol (finite state machine or automata) state transition management
- Buffer management
- Timer management
- Message management

These mechanisms are common for all communication protocols. Typically, they are designed and implemented once as a separate subsystem that comprises the set of classes, which is then used and refined on a series of projects. In this book, we will use one such subsystem, entitled the FSM library (see Chapter 6). The design and the implementation of such a library is rather specific and rests more in the domain of operating systems. Additionally, such a library frequently already exists and the designers would just use the mechanisms that it provides. Because of these two reasons, we intentionally postpone presenting the FSM library details for Chapter 4.

By accepting this approach, we keep the focus on the activities that are normally conducted during the design phase. We just assume that somebody has written the FSM library that provides all the necessary mechanisms (state transition, buffer, timer, and message management) and concentrate on the design based on these mechanisms. Therefore, for a moment we should simply think of the FSM library as an infrastructure that facilitates the design and implementation of communication protocols.

Going back to the system design itself, this chapter will cover two additional domain-specific languages that have been in use much before UML and are still rather popular today, namely SDL and MSC. The SDL diagrams are semantically equivalent to the UML activity diagrams and statecharts. In principle, establishing a one-to-one mapping between them should not be a problem. The SDL diagram, like the UML activity diagram and statechart, specifies the complete system behavior.

The SDL diagram shows states and state transitions starting from the initial state and ending in the final state. The state transitions are rendered in a style of flowcharts. Each state transition starts with an input message that causes the transition. Typically, a state transition processes the received message and optionally sends the consequent messages.

The MSC chart is semantically equivalent to the UML interaction diagrams, i.e., to both collaboration and sequence diagrams. In fact, the MSC chart can be one-to-one translated into the UML sequence diagram, but the

opposite is not the case. By looking at both of them, they make the same impression. Most engineers have the impression that they are almost the same, with the MSC being a little less expressive. Like the UML sequence diagrams, the MSC chart shows the objects that communicate—together with their corresponding vertical lines—and the messages they exchange, which are rendered as horizontal arrows connecting the source and the destination vertical lines.

Finally, this chapter covers the third domain-specific language, TTCN, which is used for making test models more formal than in UML. In contrast to the UML test model, which is rather descriptive and more like a general framework, TTCN is a well-defined language for defining test suites. As already mentioned, it originates from the ISO and has been traditionally used for the conformance testing of communication protocols.

TTCN, much like the higher-level programming language, has built-in types and allows a user to define new types (simple and structured) of variables, constraints, and functions in specialized tables. The essence of the TTCN test case specification is an indented tree of events that is filled in a table, which specifies the behavior of the testers that run the test case and the outcomes of the test case (pass, fail, or inconclusive).

The next sections describe the class diagrams (Section 3.1), the object diagrams (Section 3.2), the sequence diagrams (Section 3.3), the activity diagrams (Section 3.4), the statechart diagrams (Section 3.5), the deployment diagrams (Section 3.6), the SDL diagrams (Section 3.7), the MSC charts (Section 3.8), and the TTCN-3 test suits (Section 3.9). Chapter 3 ends with a series of design examples.

3.1 Class Diagrams

A class diagram is a special type of graph that consists of a set of vertices interconnected by arcs. They are so popular and widely used that most of the newcomers to UML equate the UML and the class diagrams. Normally, we use the class diagrams to model the static design view of the system. More precisely, we typically use them to model the vocabulary of the system, collaborations, or database schemas.

A vocabulary of the system is a set of abstractions that are parts of the system. A collaboration is a group of classes, interfaces, and other elements that cooperate to provide a more complex functionality. A schema is a blueprint that is used for the conceptual design of a database. In communication protocol engineering, we rarely deal with real databases, but we frequently need to design at least a couple of persistent objects that hold the system configuration or similar information.

The basic class diagram vertices are classes, interfaces, and collaborations. These are interconnected with three types of arcs, with dependency, generalization, and association relations. To keep the size of the class diagrams

manageable, we typically render smaller collaborations that describe certain aspects of the system. If we want to put those collaborations in a larger context, we can render the surrounding packages or subsystems. Both packages and subsystems enable hierarchical organization of class diagrams. For example, we will render the FSM library as a package that is used by the protocols that are the subjects of design and implementation.

We use packages and subsystems to manage complexity. Alternately, we render class instances (objects) in class diagrams to manage ambiguity, especially when we want to explicitly show the dynamic type of an instance or some other hidden details of the system. A special type of class diagrams are object diagrams, which will be described in the next section of this chapter.

Like use case and collaboration diagrams described in the previous chapter, class diagrams are normally also rendered using some of the commercially available graphical tools, e.g., Microsoft Visio®. The same is true for other UML diagrams described in this chapter. The basic set of graphical symbols available for rendering class diagrams is shown in Figure 3.1. The design engineer must specify properties for each instance of a symbol in the drawing.

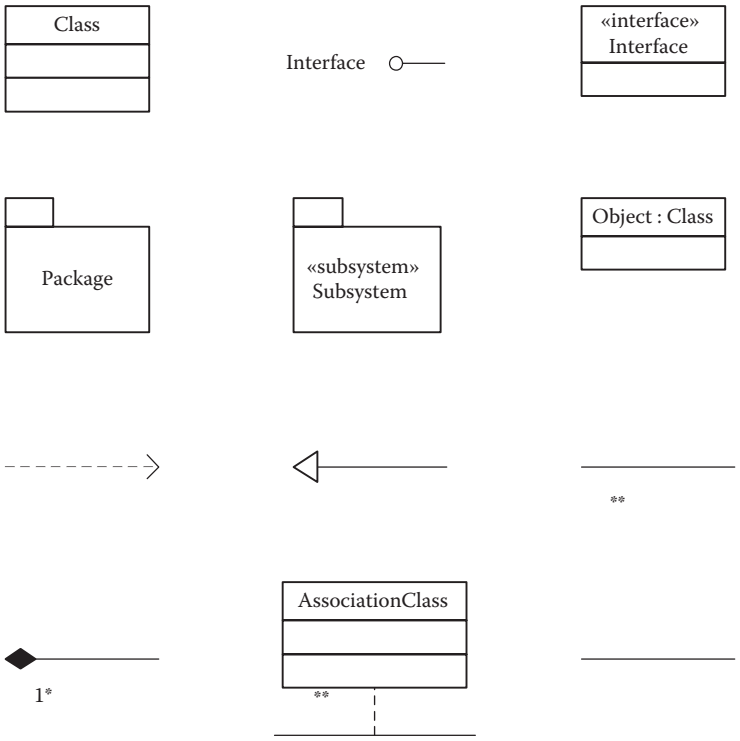


FIGURE 3.1
The basic set of graphical symbols available for rendering class diagrams.

The most frequently used symbol in class diagrams is the class symbol. Eight categories of class properties exist: the general information, the table of attributes, the table of receptions, the table of parameters, the list of components, the table of constraints, and the tagged values. The general information includes the name; the full path; the stereotype (delegate, implementation class, metaclass, structure, type, union, or utility); the visibility (private, protected, or public); and the indicators for the *Root*, *Leaf*, *Abstract*, and *Active* types of classes. The table of attributes comprises columns for the attribute name, the type, the visibility, the multiplicity (1, *, 0..1, 0..*, 1..1, or 1..*), and its initial value. The table of operations comprises columns for the operation name, the return type, the visibility, the scope (classifier or instance), and the indicator for the polymorphic operations. The table of receptions includes columns for the reception name, the corresponding signal name, the visibility, the scope, and the indicator for the polymorphic operations. The table of template parameters stores parameter names and types. The list of components comprises names of the components that implement this class. The table of constraints consists of four columns: the constraint name, the stereotype (precondition, postcondition, or invariant), the language type (OCL, text, pseudocode, or code), and the body of the constraint. The tagged values include the notes for the documentation, the location, the persistence, the responsibility, and the semantics.

Two graphical symbols are available for rendering interfaces. The first shows just the name of the interface, whereas the second also shows the available operations. Being the specialized classifier, the interface properties are a subset of class properties. More precisely, the interface has four categories of properties: the general information, the table of operations, the table of constraints, and the tagged values. Those properties are the same as the corresponding class properties with a single exception. The interface is passive in its nature, hence the general information might not include the indicator of *Active* type.

The package has four categories of properties: the general information, the table of events, the table of constraints, and the tagged values. The general information includes the name; the full path; the stereotype (facade, framework, stub, or system); the visibility (private, protected, or public); and the indicators for the *Root*, *Leaf*, and *Abstract* types of packages. The table of events stores the event names and the types.

The subsystem has four categories of properties: the general information, the table of operations, the table of constraints, and the tagged values. The general information includes the name; the full path; the visibility; and the indicators for the *Root*, *Leaf*, *Abstract*, and *Instantiable* types of subsystems.

The object has four categories of properties: the general information, the table of attributes, the table of constraints, and the tagged values. The general information about the object includes the object name and the corresponding class name. The tagged values are just documentation notes and the tag persistent value.

The dependency relation has three categories of properties: the general information, the table of constraints, and the tagged values. The general information includes the name, the stereotype (becomes, call, copy, derived, friend, import, instance, metaclass, power type, or send), and the description. The tagged values are the notes for the documentation.

The generalization relation has three categories of properties: the general information, the table of constraints, and the tagged values. The general information comprises the name, the full path, the stereotype (extends, inherits, private, protected, subclass, subtype, or uses), and the discriminator. The tagged values are documentation notes.

The association relation has three categories of properties: the general information, the table of constraints, and the tagged values (documentation notes). The general information comprises the name, the full path, the name reading direction (forward or backward), and the information about the association ends, which includes the name, the aggregation (none, composite, or shared), the visibility, the multiplicity, and the indicator *Navigable*. If the end is navigable, it is shown with an arrow symbol, and if not, it is shown without an arrow symbol. Because the composition relation is a specialization of the association relation, it has the same categories of properties (the general information, the table of constraints, and the tagged values), with the exception that the default values for the aggregation and multiplicity (of one of the ends) are composite and 1, respectively.

The association class is a class that models the complex relation; therefore, its set of properties is a union of properties of classes and associations. More precisely, the association class has five categories of properties: the general information, the table of attributes, the table of operations, the table of constraints, and the tagged values. The general information comprises the name, the full path, the information about the association ends (name, aggregation, visibility, multiplicity, and navigability), and the association class details (visibility information and *Root*, *Leaf*, *Abstract*, and *Active* indicators).

The object link has three categories of properties: the general information, the table of constraints, and the tagged values (just documentation notes). The general information includes the name and the information about each of the two link ends. The link end information comprises the name and the stereotype (none, association, global, local, parameter, or self).

This concludes the description of the basic graphical symbols available for rendering class diagrams. The usage of these symbols is illustrated by two examples, as shown in Figures 3.2 and 3.3. The first example is a simple model of the TCP/IP protocol stack, and the second example is a simple model of a finite state machine (automata).

The TCP/IP protocol stack is modeled by the classes that represent its layers: *Application*, *Transport*, *Network*, and *Interface*. The transport layer has a number of ports, which are modeled by the interface *Port*. The application depends on the transport (this fact is modeled by the dependency relation)

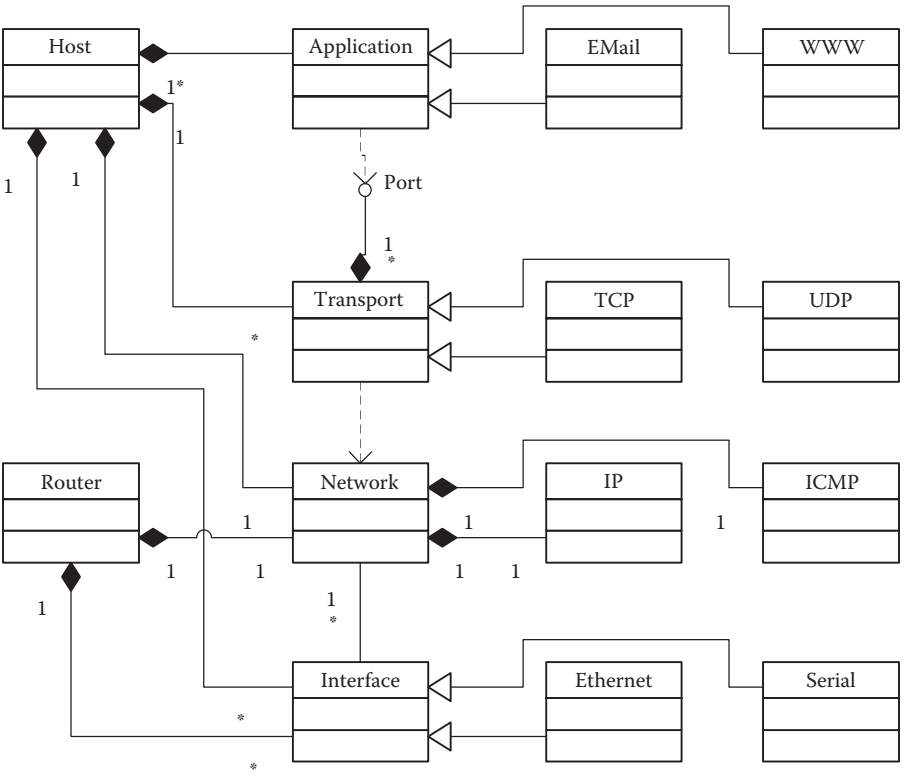


FIGURE 3.2
Example of a simple model of the TCP/IP protocol stack.

and it gets the service it needs through the interface *Port*. Further down, the transport layer depends on the network layer, which in turn is in association with a number of interfaces.

The left side of Figure 3.2 shows the models of the host computers that are connected to the Internet and the routers that interconnect the physical networks that constitute the Internet. The host computer is modeled by the class *Host*. Each host comprises all TCP/IP protocol stack layers. This fact is modeled by the composition relations between the class *Host* and the classes that model the individual layers (*Application*, *Transport*, *Network*, and *Interface*). The router is modeled by the class *Router*. Each router comprises the network and the interface layer. This is modeled by the composition relations between the class *Router* and the classes that model the individual layers.

The right side of Figure 3.2 shows some of the applications and protocols available in the TCP/IP family of protocols. The electronic mail and World Wide Web (WWW) applications—and their corresponding protocols—are modeled by the class *Email* and *WWW*, respectively. These two applications are the examples of particular applications, and this fact is modeled by the

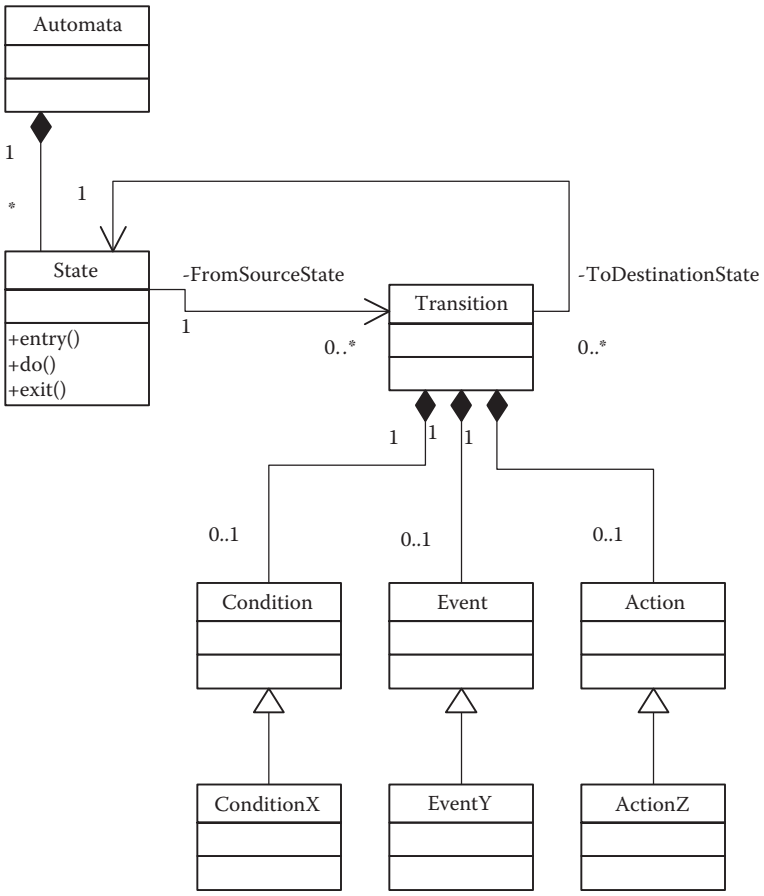


FIGURE 3.3
Example of a simple automata model.

generalization and specialization relations between the class that models a generic application (*Application*) and the classes that model the particular applications (*Email* and *WWW*). Similarly, TCP and UDP are particular transport protocols (modeled by the classes *TCP* and *UDP*), and this is modeled by the generalization and specialization relations between the class that models a generic transport protocol and the class that models TCP and UDP.

Further down the hierarchy, the Internet network layer comprises the IP and ICMP protocols (modeled as the classes *IP* and *ICMP*). This is modeled by the composition relations between the classes that model the network layer and the IP and ICMP protocols. At the bottom of the hierarchy, we show that various types of interfaces exist, e.g., Ethernet and serial, by generalization and specialization relations between the class *Interface* and the classes *Ethernet* and *Serial*, which model these particular interfaces.

The second example of simple class diagrams is a simple model of a finite state machine (automata). The aim of this example is as an easy exercise. We will return to the topic of modeling automata more comprehensively at the beginning of the next chapter. The key abstractions in this example are a finite state machine, a state, and a state transition; which are modeled by the classes *Automata*, *State*, and *Transition*, respectively (Figure 3.3).

The finite state machine comprises a number of states. This fact is modeled by the composition relation between the class *Automata* and the class *State*. The multiplicity from the side of the class *Automata* is 1 and from the side of class *State* is *. (This notation means that a finite state machine must comprise at least one state, which technically sounds like a reasonable requirement.)

The state transition links the source and the destination states, and this is modeled by two association relations between the classes *State* and *Transition*. The ends of these association relations from the side of the class *Transition* are named *FromSourceState* and *ToDestinationState*, respectively. The multiplicity from the side of the class *State* is set to 1 (because each state transition must have exactly one source and one destination state), and from the side of the class *Transition* to 0..* (because a state may have zero or more outgoing and zero or more incoming state transitions). The navigability of these two association relations is set such that the relation *FromSourceState* points from the class *State* to the class *Transition*, whereas the relation *ToDestinationState* points in the opposite direction.

The main problem with this model is ambiguity. The source and the destination states may seem to be always the same (because both *FromSourceState* and *ToDestinationState* association relations are connected to the same class, namely the class *State*). However, source and destination states can be, and most frequently are, different states. We will come back to this point shortly, after introducing additional nodes and relations available for rendering class diagrams, to resolve this problem in a less ambiguous way.

The key abstractions related to the transition are the condition that guards the transition, the event that fires the transition, and the action that is taken by the transition, which are modeled by the classes *Condition*, *Event*, and *Action*. Each transition is characterized by these three optional elements, and that is modeled by the composition relations between the class *Transition* and the classes *Condition*, *Event*, and *Action*. The fact that these elements are optional is modeled by setting the multiplicity to 0..1 from the side of the corresponding classes.

Besides actions that are taken during the transitions, we can define state bound actions, such as the action that is taken at the entrance to a certain state, the action that is performed while the system is in a certain state, and the action that is taken at the exit from a certain state (we will encounter these and more in the UML statecharts later in this chapter). These action types are modeled as the state operations *entry()*, *do()*, and *exit()*, which are defined in the table of operations for the class *State*.

Until now, we were modeling a generic finite state machine. To make this model useful for the implementation of a particular finite state machine, first we need to define the concrete conditions, events, and actions. We do so through the specialization of the base classes *Condition*, *Event*, and *Action*. Figure 3.3 shows the examples of the particular condition, event, and action, which are modeled by the classes *ConditionX*, *EventY*, and *ActionZ*, respectively. Finally, to build the particular finite state machine, we need to instantiate the classes.

This concludes the presentation of two simple examples of class diagrams. To make this graphical language more expressive and to reduce the ambiguity of the class diagrams, the graphical tool provides the additional set of graphical symbols, which are shown in Figure 3.4. The first of them is the metaclass, whose instances are classes that are added to the class diagram. We can resolve the problem of ambiguity in the previous example exactly by using the metaclass instead of the class symbol because it is then clear that the source and the destination state may both be the same state or two completely different states. Again, as for the basic set of symbols, the additional symbols have similar categories of properties. The metaclass has the same properties as the class, with the exception that its stereotype (in the general information section) is fixed to metaclass.

Both the signal and the exception symbols have the same four categories of properties, namely, the general information, the table of parameters, the table of constraints, and the tagged values. The general information is the same as for the interfaces (the name, the full path, the visibility, and the indicators *Root*, *Leaf*, and *Abstract*). The table of parameters stores the information about the parameters, which comprise the parameter name, the type, the kind (in, out, or in–out), and the default value.

The data type has five categories of properties. These are the general information, the table of enumeration literals, the table of operations, the table of constraints, and the tagged values. The general information includes the name, the full path, the stereotype (none or enumeration), the visibility, and the indicators *Root*, *Leaf*, and *Abstract*. If the data type is an enumeration, the table of enumeration literals holds the information about the literal names and the corresponding values.

A utility is a special class, therefore it has the same properties as the class with the exception that its stereotype is fixed to utility. Similarly, a parameterized class is a special class that has one or more unbound formal parameters, therefore it has the same categories of properties as the class. Related to the parameterized class is a bind relation, that binds (connects) the designated arguments to the template formal parameters. It has four categories of properties: the general information (just the name and the description), the list of bound arguments, the table of constraints, and the tagged values. The bound element adds the result of binding between the template parameters and their actual values. It has the same categories of properties as the class.

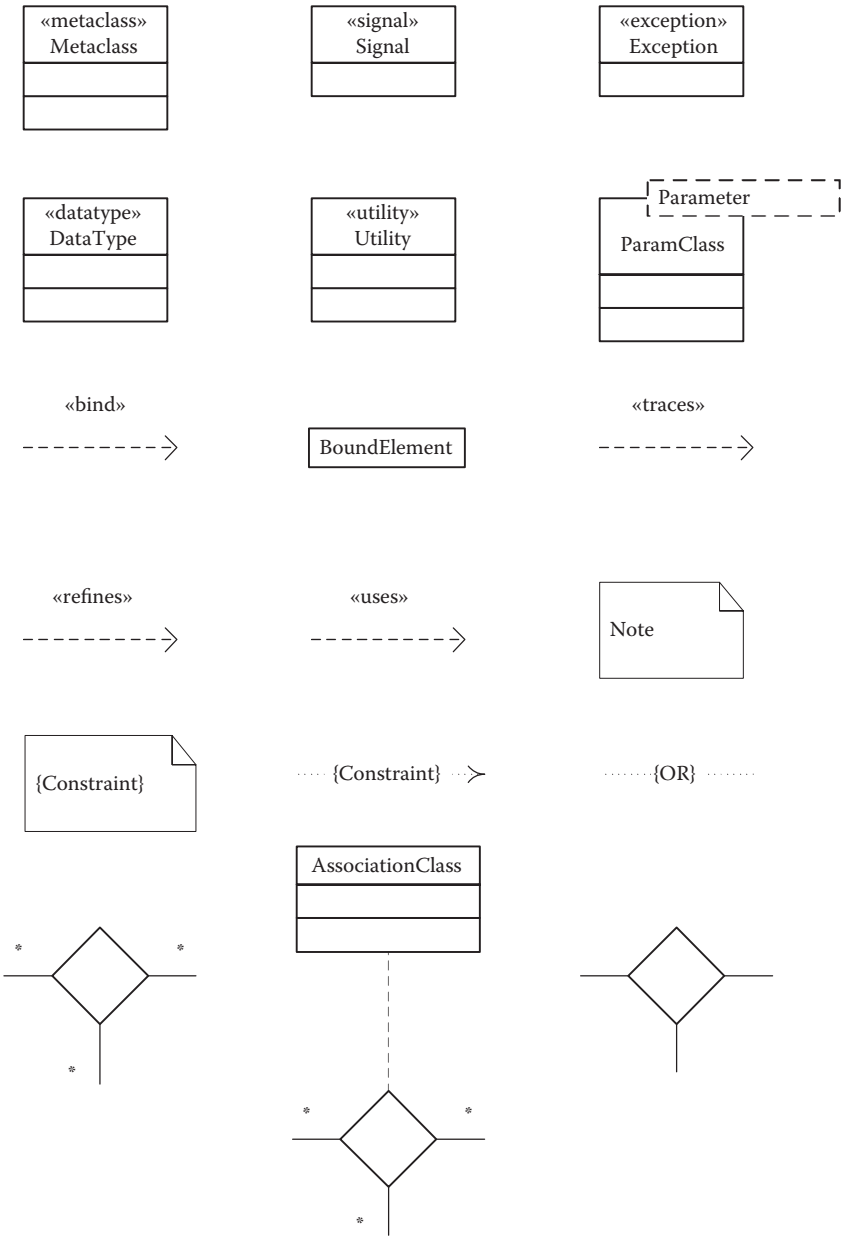


FIGURE 3.4
Additional graphical symbols available for rendering class diagrams.

The next three symbols are the traces, refines, and uses relations. We can think of them as specialized dependency relations. The traces relation connects two model elements from two different models. The refines relation connects a more detailed model element to its previous version. The uses relation indicates the dependency relationship between two model elements where one requires another to fully operate. All these relations have the same categories of information as the dependency relation, with the exception that their stereotype is fixed.

The next four symbols are the note, the constraint note, the constraint shown as arrow, and the OR constraint, which we have already encountered in both use case and collaboration diagrams (described in Chapter 2). The last three symbols are used to describe the relations among more than two model elements. The first is the *N*-ary association, which models the association among more than two classifiers. Its properties are the same as for the binary association with the additional properties for each association end (the name, the aggregation, the visibility, the multiplicity, and the navigability indicator).

The second symbol is the *N*-ary association class, which models more complex associations among more than two classifiers. Again, its properties are the same as for the binary association class with additional properties for each association end. The third and the last symbol is the *N*-ary object link, which interconnects more than two objects. Its properties are the same as the binary object link with additional properties for each end (the name and the stereotype).

At the end of this section, we focus on the domain-specific class diagrams. As already mentioned, the reader should assume and accept that somebody has already prepared the infrastructure for the design and implementation of communication protocols. There is no need to start modeling generic automata every time we start a new project, but rather we do it once and then use it on a number of projects. This practice is what in UML is called providing generic design mechanisms.

In this book, we design and implement communication protocols based on the FSM library. A typical class diagram is shown in Figure 3.5. The FSM library is shown as the package *FSMLibrary* in the diagram and, on most occasions, such representation would be sufficient. It actually comprises a rather ramified hierarchy of C++ classes (we will go into more details in the next chapter). The two most important classes are the *FiniteStateMachine* and *FSMSystem*. The fact that the FSM library contains these classes is modeled by the composition relations between the package *FSMLibrary* and the classes *FiniteStateMachine* and *FSMSystem*. The multiplicity is set to 1 on both sides (one library contains one such class).

The communication protocol is modeled by the class *Automata*. The fact that it is a specific type of finite state machine is modeled by the generalization and specialization relation between the class *Automata* and the class *FiniteStateMachine*. The former inherits all the attributes and operations from

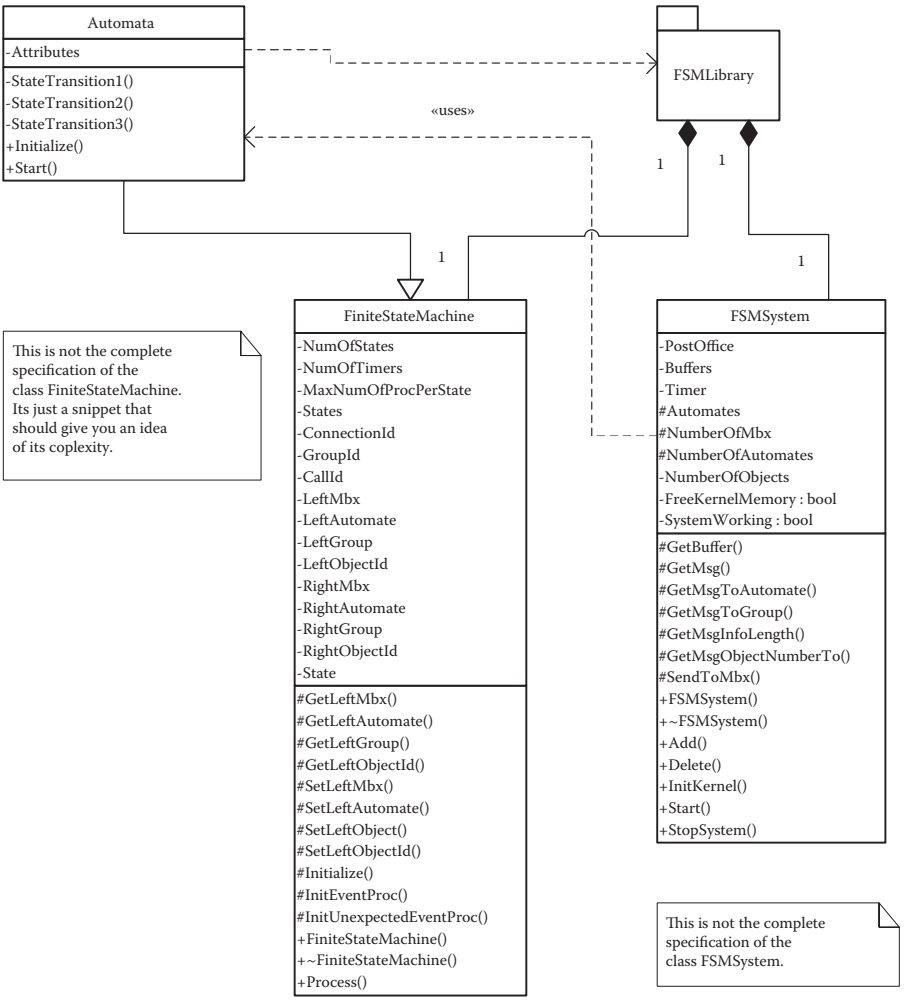


FIGURE 3.5
Typical communication protocol class diagram.

the latter. The list given in Figure 3.5 is not exhaustive, and its purpose is merely to provide the preliminary information about the basic functionality provided by the class *FiniteStateMachine*, and that it is the full set of generic design mechanisms that are needed. Once we have this class, designing a protocol essentially means defining its states and state transitions, and this is basically what we do in this chapter. After the design is finished, implementing the design (in this context) actually means writing the corresponding state transition routines (functions) in C++.

Another important class is the class *FSMSystem*. It actually provides a runtime system for all communication protocols. At the system startup, the main

program, here referred to as utility class (not shown in Figure 3.5), registers the given communication protocol by calling the method *Add()* of the class *FSMSystem*, and by giving the reference to the class that models the protocol (*Automata* in this example) as its parameter. Once registered, the protocol can receive, process, and generate events (messages) through the mailboxes provided by the *FSMSystem*.

As we will see in the next chapter, the *FSMSystem* manages all events. It analyzes the event source and destination to locate the destination protocol. Once it is found, the *FSMSystem* looks up its current state, determines the state transition routine based on the event code (type), and calls it. This mechanism is modeled by the Uses relation between the class *FSMSystem* and the class *Automata*.

As we can see, the class *Automata* is a specialization of the class *FiniteStateMachine* and is used by the class *FSMSystem* during the system run-time. More briefly stated, the class *Automata* depends on the package *FSMLibrary*. This fact is also modeled in Figure 3.5 by the corresponding dependency relation between the class and the package.

3.2 Object Diagrams

Object diagrams are a special type of class diagrams that typically show a set of objects (instances of classifiers) and their links. Pure object diagrams contain only objects and their links. However, sometimes we may put some classifiers in the object diagram, especially to clarify the relations between the classes and the objects. We may also use packages or subsystems to deal with complexity.

Object diagrams, like class diagrams, are used to show the static design view of the system. As already mentioned in the previous chapter, the collaboration diagram is used to model the behavior of the system. It also shows the architecture of the system; hence, we say that the collaboration diagram is organized by the architecture. We can think of the object diagram as one snapshot of the collaboration diagram. Imagine that time is frozen. Whatever we can see in the collaboration diagram at that single moment of time is an object diagram.

Later in this chapter, we will introduce deployment diagrams, and in Chapter 4 we will introduce component diagrams. Both deployment and component diagrams can contain only objects and their links. In such cases, they are actually pure object diagrams.

Clearly, the graphical symbols available for rendering object diagrams are the same as the symbols used for class diagrams (sometimes referred to as a static structure). In practice, we use only a very limited subset of those

symbols, most frequently only two of them (object and object link). The properties of these symbols are described in Section 3.1.

The usage of object diagrams can reduce the ambiguity of the static structure twofold. First, by rendering instances of classifiers, we can better understand the relations among them. For example, rendering just the classes in the TCP/IP protocol stack model may not give a clear indication of what the network really looks like. Second, by showing the values of the key class attributes, we can recognize reality more easily. For example, by showing the status of the individual protocols, we can comprehend their expectations from other cooperating protocols.

These ideas are illustrated by the following two examples. The first is an object diagram that shows the snapshot from a simple mail transfer protocol (Figure 3.6). The second is an example of a simple finite state machine object diagram (Figure 3.7).

Figure 3.6 shows the software running on two host computers that are connected to two different local area networks, which are interconnected by a router. The host computers clearly require full protocol stacks whereas the router requires only the two lowest level layers (IP and network interface).

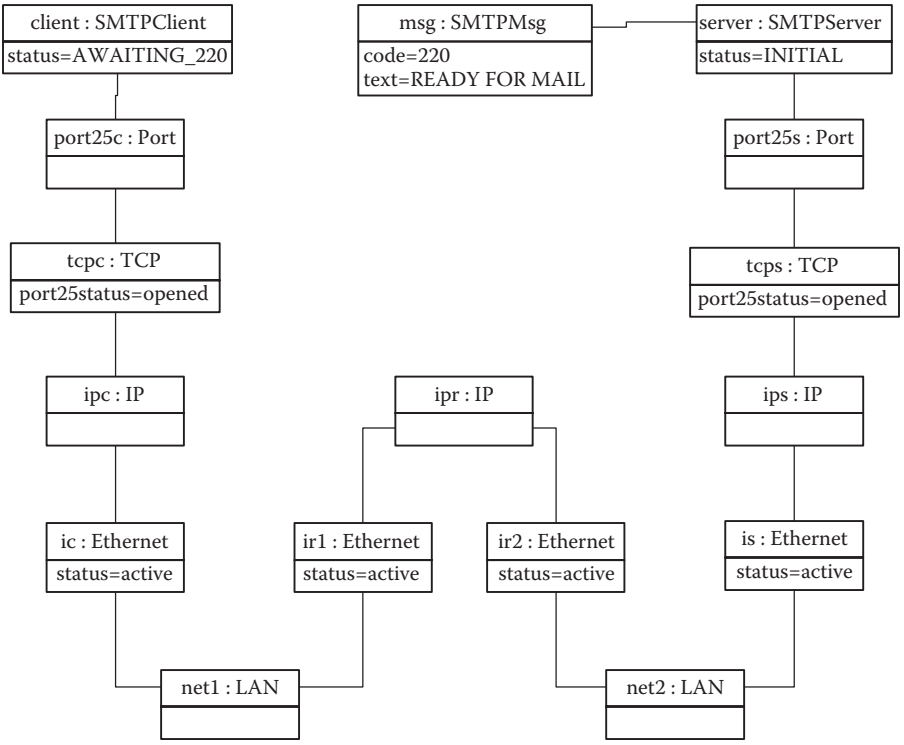


FIGURE 3.6
Snapshot from the simple mail transfer protocol (SMTP).

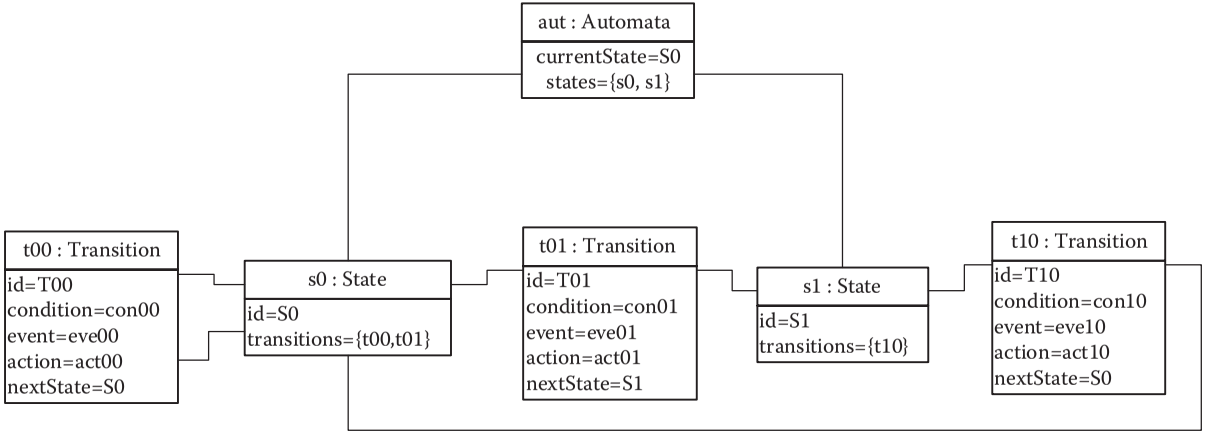


FIGURE 3.7
Example of a simple finite state machine (FSM) object diagram.

One host computer, shown on the left side of the figure, runs the SMTP client on top of the TCP/IP protocol. The other host computer hosts the SMTP server.

The first benefit of this object diagram is that it really makes clear which layers are required by the hosts and which are required by the routers. Graphically, we see the network, which was rather difficult to visualize just by looking at the class diagram shown in Figure 3.3. Enough order is found in this object diagram, too. More symbols are used than in the class diagram, but only five per host and two per router. Of course, if we try to model a large network there would be a flood of objects; therefore, we should always try to restrict our modeling to a certain aspect of a system.

The second benefit is that we can peacefully study all the details of a certain moment in the life of a protocol, in this case SMTP. It is like looking at the photograph of a certain party. This one shows the moment when the SMTP server has prepared the message 220 READY FOR MAIL and its intention was to send it at the moment when the time has been frozen. We can imagine what the sensation of looking at a series of such object diagrams would be, like watching a replica of an important event in a game in slow motion. After receiving the message 220 READY FOR MAIL, the SMTP client would prepare the message HELO, and so forth.

Besides current messages, other details are also important. For example, Figure 3.6 shows that the TCP port number 25 is opened from both sides, and from there we can deduce that the SMTP client and server had to establish the TCP connection in the first place, before they could proceed any further. Some details may seem obvious (for example, that all Ethernet cards and their drivers must be active), but they also help in making the complete picture of the selected moment. In a series of object diagrams, the changes of values of certain attributes, such as status, are the most interesting and most informative parts.

The second example of object diagrams is a simple finite state machine object diagram, which is shown in Figure 3.7. A simple finite state machine object, named *aut*, is an instance of the class *Automata* (Figure 3.4). It comprises a set of two state objects, namely *s0* and *s1*, which are the instances of the class *State*. Their identifications are *S0* and *S1*, respectively. The current state of the automata is the state with the identification *S0*.

The state object *s0* contains a set of two transition objects, namely *t00* and *t01*, which are the instances of the class *Transition* (Figure 3.4). Similarly, the state object *s1* contains a set with one transition object, named *t10*. The transition objects *t00*, *t01*, and *t10* model the automata state transitions from the state with the identification *S0* to the state with the identification *S0*, or more briefly from *S0* to *S0*, next from *S0* to *S1*, and last from *S1* to *S0*, respectively.

The attributes of the transition objects are the transition identification, the condition that guards the transition, the event that fires the transition, the action that is taken by the transition, and the next state identification. Their identifiers are *id*, *condition*, *event*, *action*, and *nextSate*, respectively. *id* and

nextState would typically be strings or integers. *condition*, *event*, and *action* are the instances of the class *Condition*, *Event*, and *Action*.

An important detail is that the values of these attributes are the instances of classes that are specialized from the classes *Condition*, *Event*, and *Action*. For example, the values of the attribute condition (namely *con00*, *con01*, and *con10*), are the instances of the classes (e.g., *Condition00*, *Condition01*, and *Condition10*), which are actually specializations of the class *Condition*. Such modeling allows us to use polymorphism, the most powerful abstraction of object-oriented design and programming.

3.3 Sequence Diagrams

Two types of UML interaction diagrams are used, namely, sequence diagrams and collaboration diagrams. We have already introduced collaboration diagrams in the previous chapter. They can be used in both the analysis and design phases of communication protocol engineering. Sequence diagrams are just another type of interaction diagrams and are semantically equivalent to collaboration diagrams. This means that a one-to-one mapping exists between these two formalisms that are used for specifying interactions.

An interaction is basically a set of objects and their relationships, together with the messages that are exchanged among the objects. Both sequence and collaboration diagrams show interactions. The major difference between them is that the sequence diagrams emphasize time ordering of messages whereas the collaboration diagrams emphasize the structural organization of a set of objects. The sequence diagrams are particularly useful for visualizing dynamic behavior in the context of the use case scenario. Generally, they are better suited for modeling sequences of events, simple iterations, and branching. Alternately, collaboration diagrams are more useful for modeling complex iterations and branching and for visualizing multiple concurrent flows of control.

Sequence and collaboration diagrams also differ in appearance. As we have already seen in the previous chapter, a collaboration diagram looks like a graph. It consists of objects that are linked together in a certain arrangement. A sequence diagram appears more like a table whose columns are related to individual objects and whose rows are related to the messages that are exchanged among the objects. We can imagine the horizontal axis x, at the top of the diagram, pointing from left to right, and the vertical axis y that points from top to bottom. The objects that participate in the interaction are arranged across the x-axis, starting on the left with the objects that are initiating the interaction and proceeding to the right with more subordinate objects. The messages that are exchanged among the objects are ordered in increasing time along the y-axis. (Actually, we have already informally

encountered sequence diagrams in Chapter 2. See the example of the SIP session setup in Figure 2.13.)

The sequence diagrams have two key features that distinguish them among other diagrams:

- Object lifeline
- Focus of control

An object lifeline is a dashed vertical line that represents the existence of an object over a period of time. The object lifeline starts with the reception of the message stereotyped as <<*create*>> and ends with the reception of the message stereotyped as <<*destroy*>>. The end of the life of an object is indicated by the mark "X." However, most of the objects will exist throughout the interaction. Such objects are normally placed at the top of the diagram and their lifeline typically goes to the end of the diagram.

The focus of control represents the period of time during which the object executes. It is rendered as a long, thin rectangle. We can model recursion, a call to self-operation, or call-back by placing a new focus of control symbol on top of the current focus of control symbol and slightly to the right, so that both of the symbols are visible. We can explicitly show the part of the focus of control where the actual computation takes place by shading the corresponding region.

We can model the mutation of objects in their state, role, or attribute values in sequence diagrams. Two methods to do this exist: The first is by placing a new copy of the object in the sequence diagram and showing the change by connecting the existing and the new object copy with the transition <<*become*>>. This procedure can be repeated if we want to show a sequence of changes. The second method is by placing a new copy of the object directly on the object's lifeline and showing the change of state, role, or attribute values then and there.

The set of graphical symbols available for rendering sequence diagrams is shown in Figure 3.8. Similar to the diagrams that were previously introduced, each of the symbols has its own properties with the exception of the focus of control, which has no properties on its own (it is a symbol that can exist only on top of the object's lifeline). The designer must fill in the properties after adding the symbol to the diagram.

The object and its lifeline have three categories of properties: the general information, the table of constraints, and the tagged values. The general information includes the name, the full path, the classifier, and the multiplicity. Other categories of properties are already explained in the previous sections.

The message has four categories of properties: These are the general information, the table of arguments, the table of constraints, and the tagged values (documentation notes). The general information includes the name, the

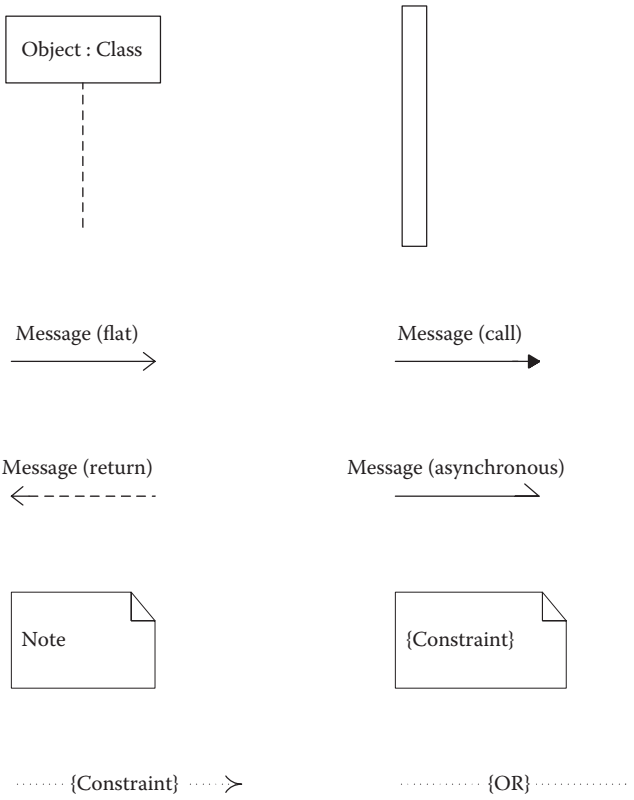


FIGURE 3.8
Set of graphical symbols available for rendering sequence diagrams.

direction (forward or backward), the operation, and the sequence expression. The table of arguments holds information about the arguments, such as the name, the type, the language, and the value.

The following four types of messages are used:

- Flat
- Call
- Return
- Asynchronous

The flat message models the communication between the objects that convey information, which should result in an action. The call message models a synchronous procedure call that should result in some action. The return message models returns from the procedure, which conveys the return value that will cause an action. The asynchronous message models the

asynchronous communication between two objects, which also carries some information that will trigger an action. The note, the constraint note, the constraint, and the OR constraint are symbols that we have already encountered and explained in Sections 3.1 and 3.2.

Next, we illustrate the use of sequence diagrams by four examples shown in Figures 3.9 through 3.12, which are semantically equivalent to the

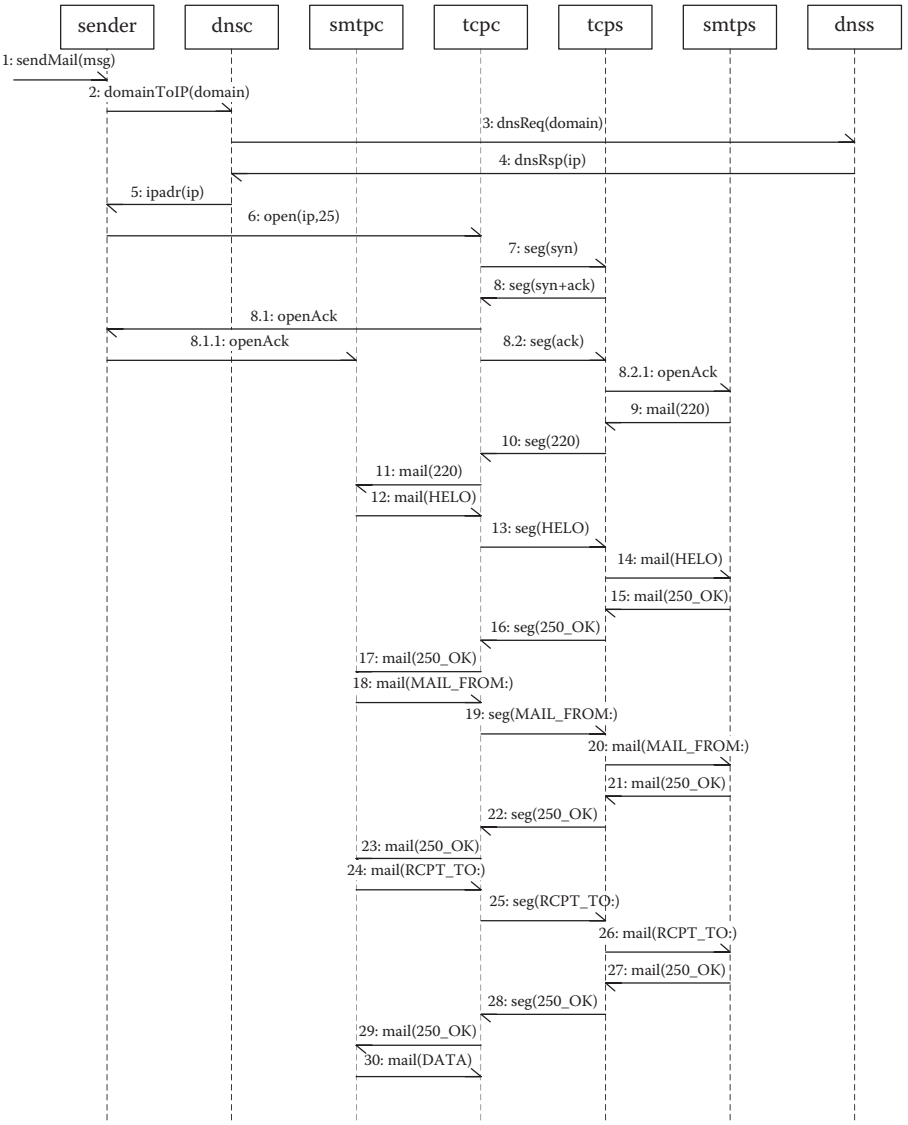


FIGURE 3.9
Sequence diagram showing the interaction between a simple program for sending and receiving e-mails and its environment.

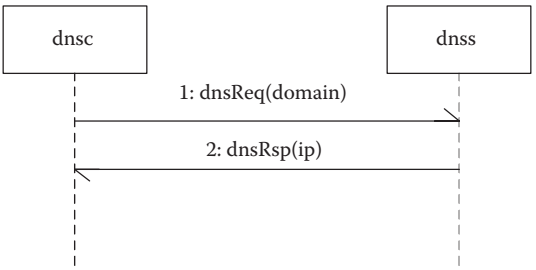


FIGURE 3.10
Sequence diagram showing the interaction between the DNS client and the DNS server.

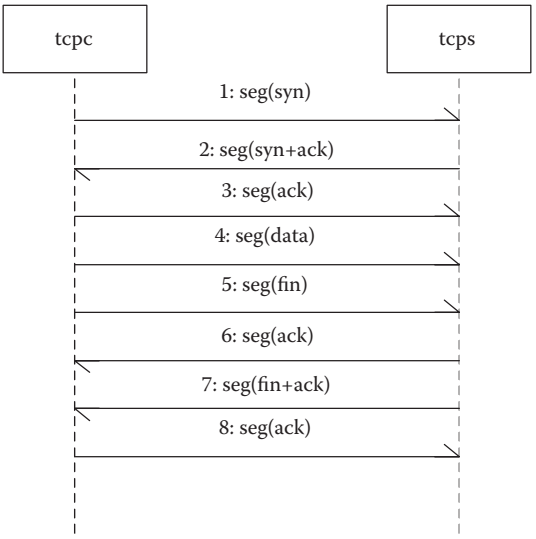


FIGURE 3.11
Sequence diagram showing the interaction between two TCP entities.

collaboration diagrams shown in Figure 2.9 through 2.12, with one exception. Figures 3.9 and 2.9 do relate to the same interaction, but they are not exactly semantically equivalent because of two reasons. First, the former shows fewer objects than the latter, mainly because of the limited diagram width. Second, the latter shows only a part of the interaction shown by the former. Interestingly enough, this seems to be a general rule. The sequence diagrams typically show fewer objects and more messages than collaboration diagrams.

The example shown in Figure 3.9 generally illustrates the same use case *Send e-mail* as the collaboration diagram shown in Figure 2.9. Figure 3.9 shows only the most important subset of objects but, at the same time, it illustrates the interaction long enough to show the moment when the SMTP client sends the SMTP message DATA toward the SMTP server. The collaboration

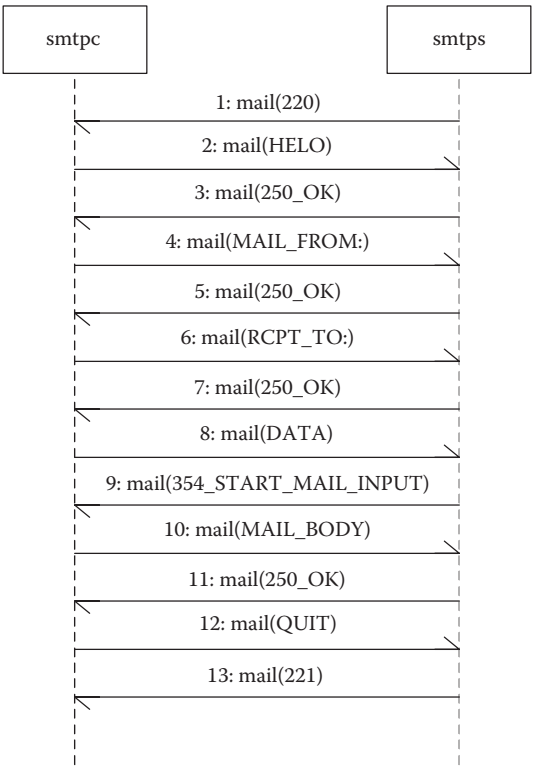


FIGURE 3.12
Sequence diagram showing the interaction between the SMTP client and the SMTP server.

diagram shown in Figure 2.9 shows the situation only up to the point when the SMTP client receives the message 220 READY FOR MAIL, which is actually the very beginning of the SMTP protocol. The names of the objects, messages (signals), and message arguments used in both figures are explained in Chapter 2. The exact flow of events shown in Figure 3.9 is as follows:

1: The object *mailc* (not shown in the diagram) sends the signal *sendMail(msg)* to the object *sender*.
2: The object *sender* sends the signal *domainToIP(domain)* to the object *dnsc*.
3: The object *dnsc* sends the signal *dnsReq(domain)* to the object *dnss*.
4: The object *dnss* sends the signal *dnsRsp(ip)* to the object *dnsc*.
5: The object *dnsc* sends the signal *ipadr(ip)* to the object *sender*.
6: The object *sender* sends the signal *open(ip,25)* to the object *tcpc*.
7: The object *tcpc* sends the signal *seg(syn)* to the object *tcps*.

8: The object *tcps* sends the signal *seg(syn+ack)* to the object *tcpc*. (The event flow now forks into two parallel flows.)

8.1: The object *tcpc* sends the signal *openAck* to the object *sender*. (The first flow begins here.)

8.1.1: The object *sender* sends the signal *openAck* to the object *smtpc*. (The first flow ends here.)

8.2: The object *tcpc* sends the signal *seg(ack)* to the object *tcps*. (The second flow begins here.)

8.2.1: The object *tcps* sends the signal *openAck* to the object *smtps*.

9: The object *smtps* sends the signal *mail(220)* to the object *tcps*. (Note: We have restarted the message numbering here for brevity. We promoted 8.2.2 to 9.)

10: The object *tcps* sends the signal *seg(220)* to the object *tcpc*.

11: The object *tcpc* sends the signal *mail(220)* to the object *smtpc*.

12: The object *smtpc* sends the signal *mail(HELO)* to the object *tcpc*.

13: The object *tcpc* sends the signal *seg(HELO)* to the object *tcps*.

14: The object *tcps* sends the signal *mail(HELO)* to the object *smtps*.

15: The object *smtps* sends the signal *mail(250_OK)* to the object *tcps*.

16: The object *tcps* sends the signal *seg(250_OK)* to the object *tcpc*.

17: The object *tcpc* sends the signal *mail(250_OK)* to the object *smtpc*.

18: The object *smtpc* sends the signal *mail(MAIL_FROM:)* to the object *tcpc*.

19: The object *tcpc* sends the signal *seg(MAIL_FROM:)* to the object *tcps*.

20: The object *tcps* sends the signal *mail(MAIL_FROM:)* to the object *smtps*.

21: The object *smtps* sends the signal *mail(250_OK)* to the object *tcps*.

22: The object *tcps* sends the signal *seg(250_OK)* to the object *tcpc*.

23: The object *tcpc* sends the signal *mail(250_OK)* to the object *smtpc*.

24: The object *smtpc* sends the signal *mail(RCPT_TO:)* to the object *tcpc*.

25: The object *tcpc* sends the signal *seg(RCPT_TO:)* to the object *tcps*.

26: The object *tcps* sends the signal *mail(RCPT_TO:)* to the object *smtps*.

27: The object *smtps* sends the signal *mail(250_OK)* to the object *tcps*.

28: The object *tcps* sends the signal *seg(250_OK)* to the object *tcpc*.

29: The object *tcpc* sends the signal *mail(250_OK)* to the object *smtpc*.

30: The object *smtpc* sends the signal *mail(DATA)* to the object *tcpc*.

Another practical detail about sequence diagrams is that not only their width but also their height is limited. Because of this, we are normally forced

to break the flow of events at a certain point. In the previous example, it was after the object *smtpc* has sent the signal *mail(DATA)* to the object *tcpc*. Typically, we would continue that flow on another sequence diagram. A good practice is to pick the breaking points logically, for example, at the beginning or at the end of certain communication phases.

It is also important to emphasize that the sequence diagram in Figure 3.9 shows only main flows of events. It does not show what happens in the case of errors. The error handling is typically shown in separate sequence diagrams. We can use packages to wrap together all the related sequence diagrams.

Figure 3.9 shows also that the real overall interaction can be fairly complex. To deal with the complexity, we can focus on the individual virtual interactions instead. For example, the sequence diagram showing the interaction between the DNS client and server is a trivial one (Figure 3.10). The overall flow of events is then reduced to only the following two events:

1: The object *dnsc* sends the signal *dnsReq(domain)* to the object *dnss*.
2: The object *dnss* sends the signal *dnsRsp(ip)* to the object *dnsc*.

Similarly, the virtual interaction between two TCP entities, modeled by the objects *tcpc* and *tcps*, is governed by the TCP protocol. It is slightly more complex and comprises the following flow of events (Figure 3.11):

1: The object *tcpc* sends the signal *seg(syn)* to the object *tcps*.
2: The object *tcps* sends the signal *seg(syn+ack)* to the object *tcpc*.
3: The object *tcpc* sends the signal *seg(ack)* to the object *tcps*.
4: The object *tcpc* sends the signal *seg(data)* to the object *tcps*. (This is the data transmission phase.)
5: The object *tcpc* sends the signal *seg(fin)* to the object *tcps*.
6: The object *tcps* sends the signal *seg(ack)* to the object *tcpc*.
7: The object *tcps* sends the signal *seg(fin+ack)* to the object *tcpc*.
8: The object *tcpc* sends the signal *seg(ack)* to the object *tcps*.

Finally, the virtual interaction between the SMTP client and server, modeled by the objects *smtpc* and *smtps*, is of the same order of complexity (Figure 3.12). The interaction is governed by the SMTP protocol. The corresponding flow of events is the following:

1: The object *smtps* sends the signal *mail(220)* to the object *smtpc*.
2: The object *smtpc* sends the signal *mail(HELO)* to the object *smtps*.
3: The object *smtps* sends the signal *mail(250_OK)* to the object *smtpc*.
4: The object *smtpc* sends the signal *mail(MAIL_FROM:)* to the object *smtps*.

5: The object *smtps* sends the signal *mail(250_OK)* to the object *smtpc*.
6: The object *smtpc* sends the signal *mail(RCPT_TO:)* to the object *smtps*.
7: The object *smtps* sends the signal *mail(250_OK)* to the object *smtpc*.
8: The object *smtpc* sends the signal *mail(DATA)* to the object *smtps*.
9: The object *smtps* sends the signal *mail(354_START_MAIL_INPUT)* to the object *smtpc*.
10: The object *smtpc* sends the signal *mail(MAIL_BODY)* to the object *smtps*.
11: The object *smtps* sends the signal *mail(250_OK)* to the object *smtpc*.
12: The object *smtpc* sends the signal *mail(QUIT)* to the object *smtps*.
13: The object *smtps* sends the signal *mail(221)* to the object *smtpc*.

3.4 Activity Diagrams

Up to now, we have introduced three types of diagrams that are used for modeling dynamic aspects of systems. These are the use case, the collaboration, and the sequence diagrams. The use case diagrams are used first for capturing the requirements of the system. Next, they are translated into collaboration diagrams that model the architecture of the system. Then, at the beginning of the design phase, both collaboration and sequence diagrams are used for building up the storyboards of scenarios.

These scenarios describe the interaction among the most interesting objects; hence, we refer to them as interaction diagrams. The interaction itself is shown by the messages that are dispatched among the objects. Generally, interaction (collaboration and sequence) diagrams are similar to Gantt charts. The main difference between the collaboration and sequence diagrams is that the former emphasizes structural relations whereas the latter emphasizes the time ordering of messages.

The storyboards of scenarios are a good place to start the design—therefore, they are a type of design front-end. Although the interaction diagrams make a perfect start of the design, they are seldom used as the final artifacts of the design phase because of two problems:

- The interaction diagrams are most frequently incomplete.
- The interaction diagrams specify the external behavior of individual objects, leaving their internal behavior unknown.

As already mentioned, the interaction diagrams typically cover the main flow of events and, because of the limited space in the diagrams, even the

main flow must be partitioned into logical communication phases. Other, less frequent flows (including error handling) are modeled in additional interaction diagrams. All these diagrams can be sorted into packages for easier manipulation. However, no matter how pedantic the engineer is, the set of interaction diagrams remains incomplete by an unwritten rule. Some scenarios are always missing. In the area that is of primary interest for this book, the packages of interaction diagrams are especially vulnerable to the specification of timers and complex, unforeseen error scenarios.

Another problem we encounter while trying to make the packages of interaction diagrams complete is that they become voluminous and, as a result, hard to comprehend. This behavior is what we should expect when we try to enumerate and describe the cases instead of trying to create the rules that generate these cases. Even a simple program performing some simple arithmetic calculations can produce enormous numbers of execution cases when we take into account the cardinal numbers of sets of values that the common variable types can have. Because of the coverage problems, an implicit engineering rule is that a design based solely on the interaction diagrams is considered as incomplete. This may not be true in the case of simple systems, but generally it is. Therefore, we need the design back-end: the means to end the design.

The secret of how to finish the design is found by turning our attention to the internal behavior of the objects and trying to specify it. This attitude is like turning the interaction diagrams inside out. We want to specify the activities that should take place to provide the desired external behavior and what should be the order (flow) of the activities in the scope of a single object or in the scope of a set of objects that are involved in the interaction. The means to do this in UML are the activity diagrams, which are similar to PERT network charts. The alternative means to specify the behavior of single objects in UML are statecharts, which will be introduced in the next section.

An activity diagram is essentially a flowchart that shows the flow of control from activity to activity. If we model the behavior of a single object, we render the flow of control within that single object. The activity diagrams are even more powerful and they allow us to model the behavior of a group of objects by rendering the flow of control in that larger scope. Additionally, we can model a single flow of control or more concurrent flows of control within both a single object and a group of objects.

In the context of a single object, we typically partition its behavior into a set of its operations and then model the flow of control of these operations individually. Therefore, the most elementary level of modeling by using activity diagrams is the level of the object's operation. On the opposite side of the scope scale, we can model the workflow of a group of cooperating objects. We will return to that point shortly.

The most elementary activity is an **action state**. It is defined as an atomic (i.e., uninterruptible) program computation. Examples of action sates are the following:

- Create another object
- Destroy another object
- Call an operation on an object
- Return a value
- Send a signal to an object
- Receive a signal from an object
- Evaluate an expression
- Execute a single statement

The action states can be specified in informal text, pseudocode, or a higher-level programming language. Although it is generally assumed that the action state takes a small amount of execution time, that finite amount of time must be taken into account, especially in the models of hard, real-time systems.

By combining more action states, we are building more complex activities, which are referred to as **activity states**. We can think of the activity state as a composite state that is made of other activity states and action states. The activity state can also comprise some special actions, such as entry and exit actions. The former is taken at the entrance to the activity state, and the latter is taken at its exit.

The state transitions in activity diagrams normally take place after completion of the last activity in the originating state. A transition without a guard (condition) immediately passes control to the destination state. Such a transition is referred to as a triggerless, or completion, transition. A transition can branch into two or more guarded transitions, or it can fork into more concurrent transitions. More concurrent transitions can join into a single transition, as we will explain shortly with some simple examples.

An activity diagram is a special type of a graph that comprises a set of vertices that are interconnected by arcs. The basic set of graphical symbols available for rendering activity diagrams is shown in Figure 3.13. Each symbol has a set of properties that must be set by the designer once they add a symbol to the diagram.

The initial state has three categories of properties. These are the general information, the table of constraints, and the tagged values (documentation notes). The general information is just the name and the type (initial). Each activity diagram must start with this symbol.

The final state has the same categories of properties as the initial state symbol, with the exception that its type is final. If the activities specified by the activity diagram go on forever, the diagram will not contain this symbol. Alternately, it can contain one or more such symbols.

The action state has five categories of properties, namely, the general information, the call action, the list of deferred events, the table of constraints, and the tagged values (documentation notes). The general information comprises

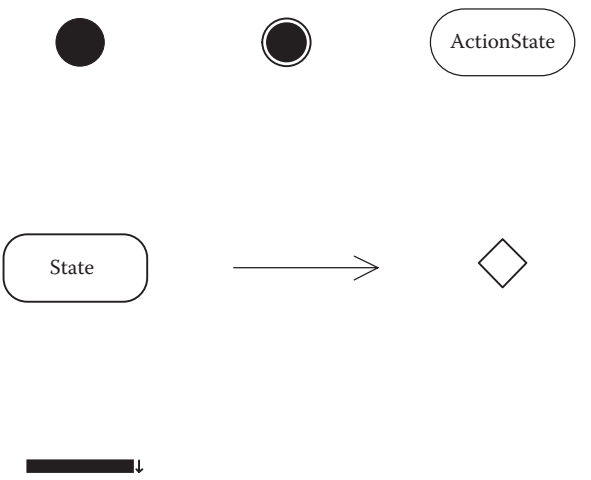


FIGURE 3.13
The basic set of graphical symbols available for rendering activity diagrams.

the name, the stereotype, and the partition. The call action specifies the name of the operation and the table of its arguments, which holds information about the argument name, type, language, and value.

The activity state has six categories of properties. These include the general information, the table of entry actions, the table of exit actions, the table of internal transitions, the table of constraints, and the tagged values. The general information is just the name and the stereotype. Both the table of entry and the table of exit actions store the corresponding action names and their types. The table of internal transitions comprises their properties. Each internal transition is characterized by its name, its stereotype, and the event that triggers the transition.

The control flow transition has four categories of properties, including the general information, the table of actions, the table of constraints, and the tagged values (documentation notes). The general information comprises the name and, optionally, the corresponding event and the guard expression. The table of actions holds action names and their types. The decisions, as well as the fork and join transitions, have three categories of properties, namely, the general information (just the name), the table of constraints, and the tagged values.

We illustrate the usage of these basic symbols by the following four simple examples shown in Figures 3.14 through 3.17. The example in Figure 3.14 shows a simple sequence of interruptible activities (i.e., activity states), namely, *openPort(p)*, *sendData(seg)*, and *closePort(p)*. Normally, these activity states would be modeled by the activity diagrams themselves on the subordinated level of the hierarchy. The control flow transitions between the individual activity states in this example are triggerless, or completion

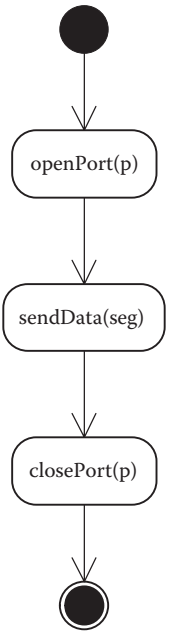


FIGURE 3.14
An example of a simple sequence of activity states.

transitions, which means that they are not triggered by other events. They also may not be guarded because their sources are not decisions.

The exact semantics of the states in this example are not really important; for example, we can interpret it as open the given port, send the given segment of data, and close the port at the end. Generally, we should think of the activity state as an operation (i.e., procedure or function) which consists of executable statements or calls to other operations, including calls to itself (recursion). Thinking about forward engineering helps make useful activity diagrams. Try to imagine how the model would map to the code. It really does not make any difference how the mapping is made, either automatically with a tool or by hand.

The example in Figure 3.15 is an illustration of activity flow with branching. Actually, it is a simplified implementation of the reliable transport mechanism known as Automatic Repeat Question (ARQ). The whole operation begins by starting the retransmission timer *T1*. This beginning is modeled by the activity state *startTimer(T1)*. The operation then sends the datagram and waits for the answer. These two activities are modeled by the activity state *sendPacket(d)* and *a=waitAnswer()*, respectively.

If the retransmission timer expires, the packet is retransmitted. This mechanism is modeled by the transition guarded by the expression *[T1 expired]*, the activity state *restartTimer(T1)*, and the completion transition back to the activity state *sendPacket(d)*. The reception of the answer is modeled by the

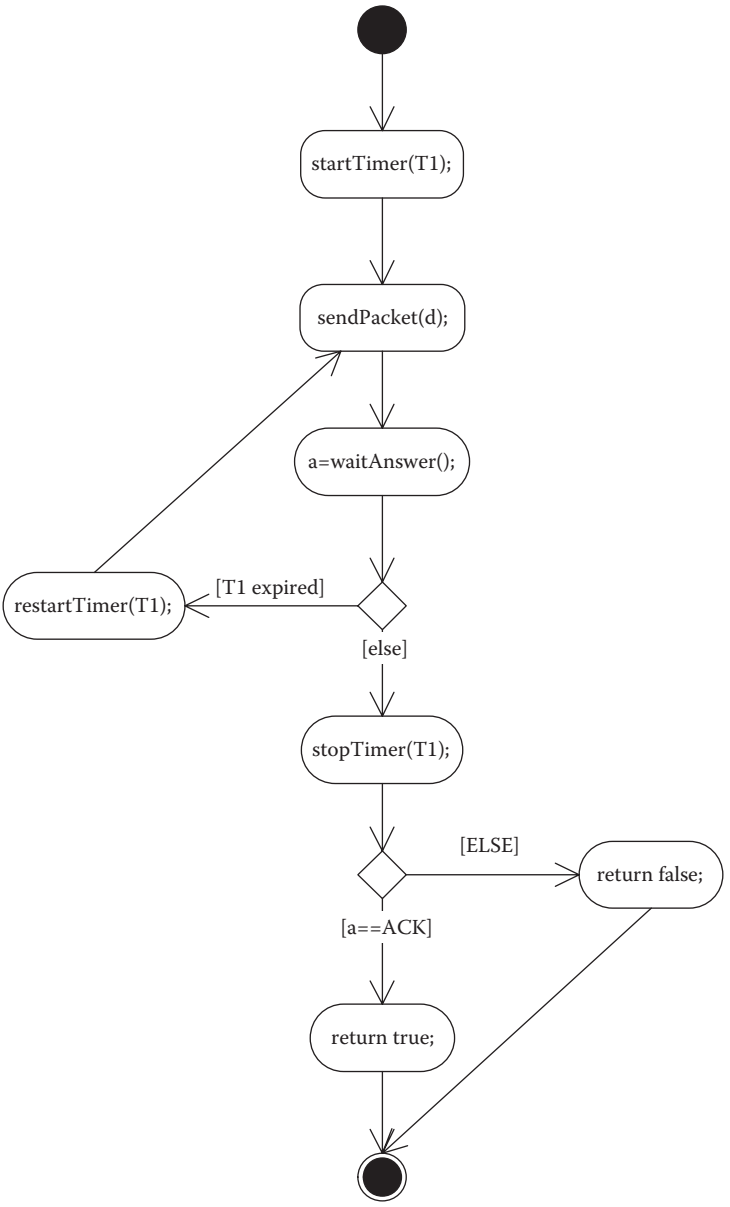


FIGURE 3.15
Example of a simple flow of activities with branching.

transition that covers all the other cases (guard expression *[ELSE]*). The operation proceeds by stopping the retransmission timer, and this action is modeled by the activity state *stopTimer(T1)*. If the answer is the acknowledgment (*ACK*), the operation returns the *value true*; otherwise, it returns the *value false*.

The previous example uses two branches. Each branch has one incoming and two or more outgoing transitions. The outgoing transitions are guarded by the Boolean expressions that are evaluated at the entrance to the branch. The set of guards has two important features:

- The guards must not overlap—this makes the flow of control unambiguous.
- The guards must cover all possibilities—this ensures that the flow of control is not going to freeze.

Precisely these two features force us to make complete models and specifications of activities that describe the behavior of the system. When we render interaction (collaboration and sequence) diagrams, no such enforcements are present. As a result, they remain unfinished. Of course, at the time when we render interaction diagrams, we really do not want to make them final; rather, we want to check the most important aspects and scenarios, and to make our analysis more comprehensive and useful for the finalization later. Therefore, when we start rendering the activity diagram, we already have a good overall vision, but non-overlapping and complete coverage features are the driving forces of the design finalization.

One safe way to provide both of these features is to use only the decisions with two outgoing transitions and to guard one of them by the keyword *ELSE*, as in the example in Figure 3.15. Special attention should be paid to the decisions with more outgoing transitions, which are guarded by explicit expressions (i.e., without the keyword *ELSE*). However, the price that we may pay for safety is ambiguity. For example, if the operation in the previous example returns the *value false*, it might do so because the correct not acknowledge answer (*NAK*) has been received. However, the operation will return the same value if any other message (including corrupted *ACK* or *NAK*) has been received.

The example in Figure 3.16 illustrates the usage of loops in activity diagrams. Imagine that the IP protocol must route a datagram over a physical network, which has the Maximal Transfer Unit (MTU) smaller than the datagram size. Normally, the IP protocol partitions the datagram into fragments (that fit MTU) and routes the resulting fragments individually in such cases. The standard means to model repetitive activities in activity diagrams are loops.

The example in Figure 3.16 starts by setting the control variable i to the value 0. It continues with no operation activity state, followed by the decision

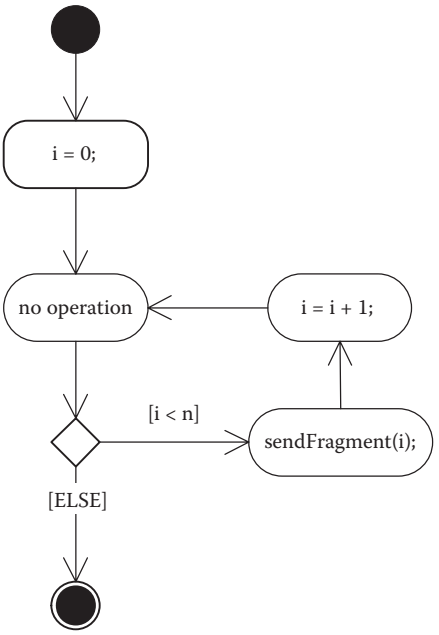


FIGURE 3.16
Example of a loop in an activity diagram.

that checks the loop continuation condition ($i < n$). If the condition is satisfied, the flow enters the loop body (*sendFragment(i)*). The loop body is followed by the activity state that updates the control variable ($i = i + 1$). The example terminates when the loop continuation condition becomes false.

The example in Figure 3.17 shows the usage of concurrent control flows. Imagine that we want to model a simple communication over the TCP connection. First, we must establish the TCP connection by opening a particular TCP port. We model this by the activity state *openPort(p)*. Once the connection is established, the TCP protocol provides simultaneous transfer of data in both directions (full-duplex). To model that, we need to fork a single flow of control into two parallel (concurrent) flows of control. One of them enters the activity state *sendData()*, which models the activity of sending the data to the remote site. The other control flow enters the activity state *receiveData()*, which models the activity of receiving the data from the remote site.

These two activities logically evolve in parallel over time. On a multiprocessor system, they can be deployed on two different processors to maximize the system throughput. In such a case, these two activities would also be parallel in reality. Alternately, single-processor systems create quasi-parallelism using the time-sharing operating system. The activities are then not parallel in reality, but they are still concurrent because they can compete for the same resources. Additionally, the activities can communicate using signals.

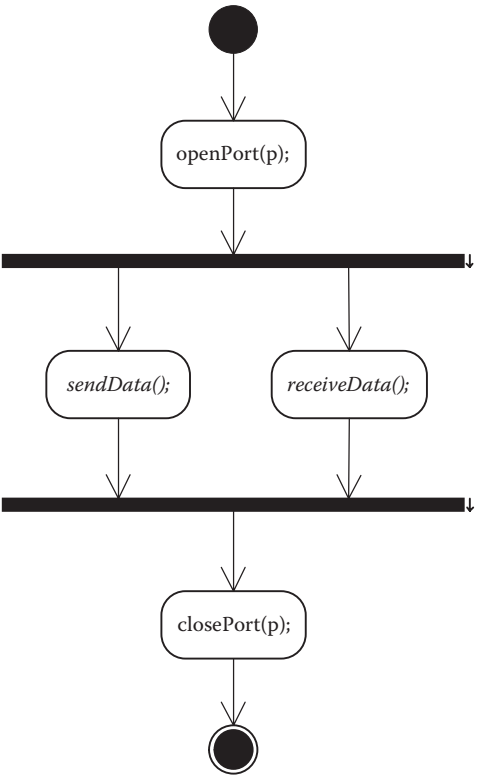


FIGURE 3.17
Example of a simple set of concurrent flows.

Traditionally, such communicating sequential processes are referred to as **coroutines**.

Although the model shown in Figure 3.17 is fairly simple, it may reflect a realistic communication, such as a Telnet session. Imagine that the activity state *sendData()* is a composite state that reads the user keystrokes and sends them to the Telnet server over the TCP connection, in a loop, until the end-of-file key combination is detected. The activity state *receiveData()* in this scenario would be also a composite activity state, which receives the responses from the Telnet server and displays them on the monitor in a loop, until the end-of-communication signal is detected (typically, it would be sent when the end-of-file key combination is detected).

Once one of the parallel activities finishes, it proceeds to the control flow joint synchronization point where it waits for the other parallel activity to finish. When both of the activities are finished, the corresponding parallel control flow joins into a single control flow, which enters the activity state *closePort(p)*; after finishing that activity, it terminates.

As we have seen from the previous example, fork and join synchronization points are rendered as either thick horizontal or vertical lines. It is important to remember that they must be balanced. Similar to the subexpression—which must begin with the opening parenthesis and end with the closing one—each nesting level of the concurrent control flows must begin with the fork symbol and end with the corresponding join symbol. Apart from that, no restrictions are placed on the number of nesting levels, at least not in theory. Of course, in practice we should not go beyond a manageable number.

The set of additional symbols that are available for rendering activity diagrams is shown in Figure 3.18. These are the object in state, the object in flow, and the swim lane symbols, as well as the symbols common for all diagrams, namely, the note, the constraint note, the two-element constraint, and the OR constraint.

The object flow transition enables us to show how the object state changes in the activity diagrams. Typically, we render the objects showing the current and the new states and we connect them by the object flow transition. The objects themselves may be results of activity states and can be used by other activity states. The object flow symbol has the same four categories of properties as the control flow symbol (described previously in this section).

The swim lane has no strict semantics. It is normally used to show individual parties in the workflows. The swim lane is typically implemented as a class or a set of classes. It is better suited for modeling business processes, but it can also be used for modeling communication protocols. The swim lane has three categories of properties: general information (essentially, its name), the table of constraints, and the tagged values.

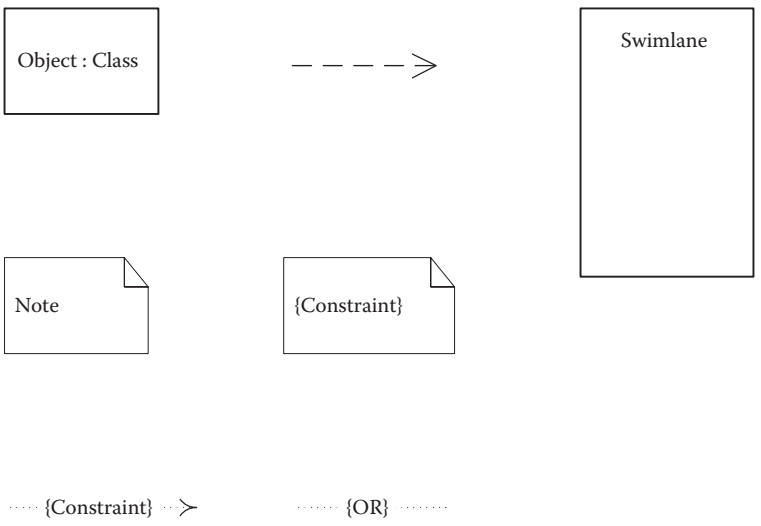


FIGURE 3.18
Additional graphical symbols available for rendering activity diagrams.

The example in Figure 3.19 illustrates the usage of objects, data flow transitions, and swim lanes, with the example of activities initiated by the Domain Name System (DNS) client request for mapping a given domain name onto the corresponding IP address. Figure 3.19 is a type of a workflow conducted by the DNS client and server in their cooperative work of translating a domain name into the IP address. The DNS client is represented by the first swim lane and the DNS server is represented by the second. This activity

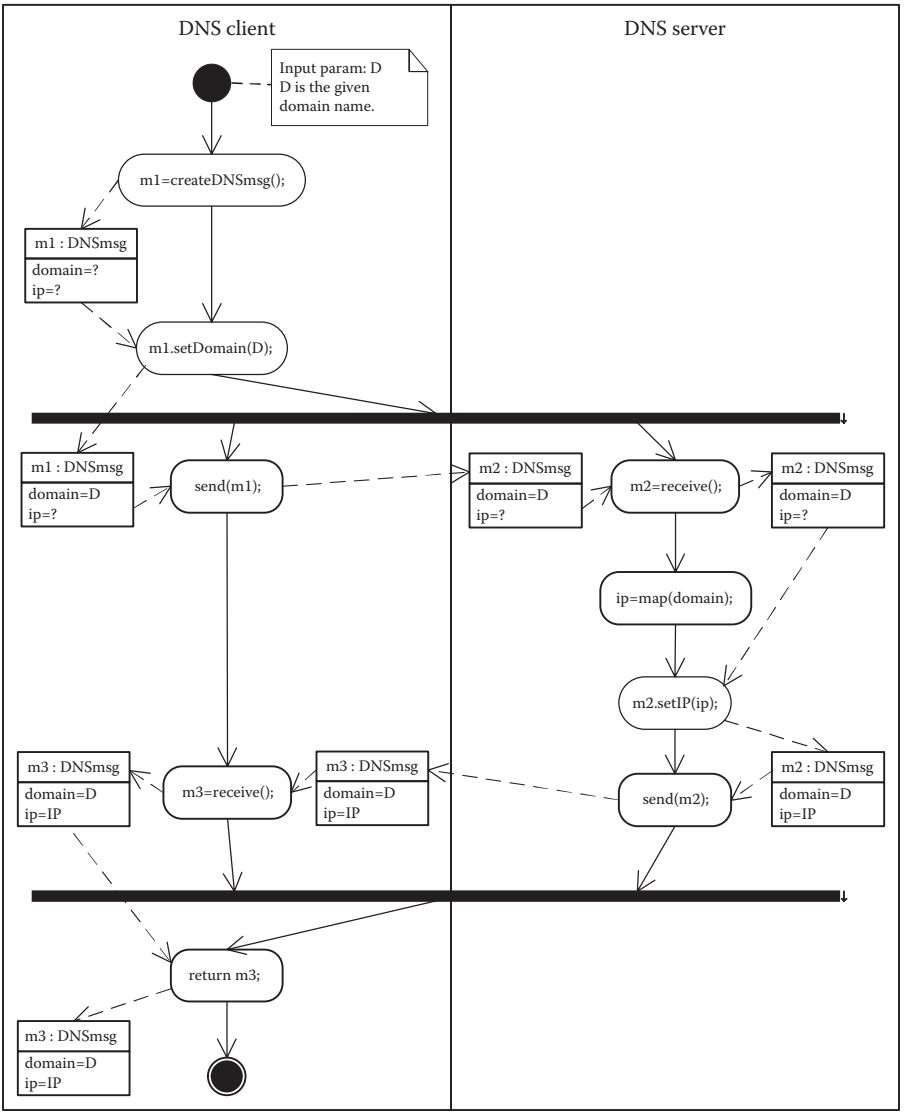


FIGURE 3.19
Workflow between the DNS client and server with the message flow.

diagram shows both the control flow among individual activity states and data flow, which are created by a series of objects that are consumed and produced by the activity states of both DNS client and server.

The given domain name is the input parameter of the DNS client operation that translates the domain name into the corresponding IP address. This operation starts by the activity state *createDNSmsg()*, which creates an empty DNS message. This action is modeled by placing the object *m1* that represents the DNS message in the activity diagram and by connecting it to the activity state *createDNSmsg()*, with the arrow pointing toward the object *m1*. This means that the object *m1* is produced by the activity state *createDNSmsg()*. The fact that the message is empty is indicated by the unknown values of both attributes *domain* and *ip* (the unknown value is denoted by the question mark character, "?").

Next, the activity state sets the attribute domain to the value of the input parameter *D*, thus creating a new state of the object *m1*. This action is modeled by placing a new copy of the object *m1* in the activity diagram and by adding two object flow arcs. The first connects the previous object copy and the activity state *m1.setDomain(D)*. The arrow points toward the activity state, which means that the state consumes the object. The second object flow arc connects the activity state and the new copy of the object *m1*, thus implying that the activity state produces it.

The control flow then forks into two independent flows. One is conducted by the DNS client and the other is conducted by the DNS server. The DNS client continues by sending the DNS message as a DNS request to the DNS server. The corresponding activity state creates a new object, named *m2*, and places it in the second swim lane, because we assume that the DNS server runs on a different machine, or at least in a different address space. The DNS server, in turn, receives the DNS message. A common mechanism for copying the message from an internal operating system buffer to the buffer that is located within the address space of the DNS server is modeled by placing two different copies of the object *m2*.

The DNS server continues by translating the given domain name into the corresponding IP address and by setting the attribute *ip* to the value IP, which denotes the result of that translation. This fact is shown in the third copy of the object *m2*. The DNS server proceeds by sending the completed DNS message, which models the DNS response message, to the DNS client, which, in turn, receives it and creates the copy of the object *m3* in its address space. Finally, two independent control flows join together and the DNS client returns the completed DNS message to its user, thus creating the final copy of the object *m3*.

As this example shows, the models of the workflows are useful because they show and specify the external behavior, i.e., the interface and protocol between the objects in the form of the corresponding sequence of messages exchanged by the objects, as well as the internal behavior of objects in the form of the series of activity states visited by them. The first is created by

modeling the data and object flow, and the second is created by modeling the control flow across the objects. Again, by taking care of the complete coverage of possibilities, without any overlaps, we ensure that the model is complete. (This was not the main goal of the last example, at least not to the extent of the previous one, but we should keep that in mind.)

Figure 3.20 shows the activity diagram for one real protocol, TCP, and follows the conventions introduced by the corresponding IETF RFC 793. The user requests are written in capital letters. The user requests are *OPEN*, *SEND*, and *CLOSE*. Two types of *OPEN* requests are used, namely active *OPEN* and passive *OPEN*. The difference between the two depends on which one is taking the initiative in the connection establishment procedure.

The next convention is that the names of the events and actions are written in lowercase letters, with the following abbreviations:

- TCB (Transmission Control Block)
- snd (send)
- rcv (receive)
- SYN (indicates that the synchronization bit of the TCP segment is set)
- ACK (indicates that the acknowledgment bit of the TCP segment is set)
- SYN, ACK (both SYN and ACK bits are set)
- FIN (indicates that the final bit of the TCP segment is set)
- ACK of SYN (denotes the acknowledgment of the SYN segment)
- ACK of FIN (denotes the acknowledgment of the FIN segment)
- MSL (Maximum Segment Lifetime)

The TCP events are actually modeled as guard expressions whereas the TCP activities are modeled as UML action states (a relatively short and uninterruptible series of executable statements). Notice that we could model the TCP activities either by action or by activity states because these activities are essentially interruptible. However, because they can be implemented as rather short routines—which do not involve reception of any signals—modeling them as action states makes more sense than as activity states.

The TCP protocol spends most of the time in one of its stable states waiting for a certain event to occur. The TCP stable states are modeled by the UML activity states. While being in one of its stable states, the TCP protocol just waits for an event (it does not execute any statements). The process that executes the TCP protocol is blocked and it does not compete for the processor's execution time. Therefore, the activity corresponding to the stable state is more than interruptible—it is blocked. Because such an abstraction is missing in the UML activity diagrams, we are forced to model it with an

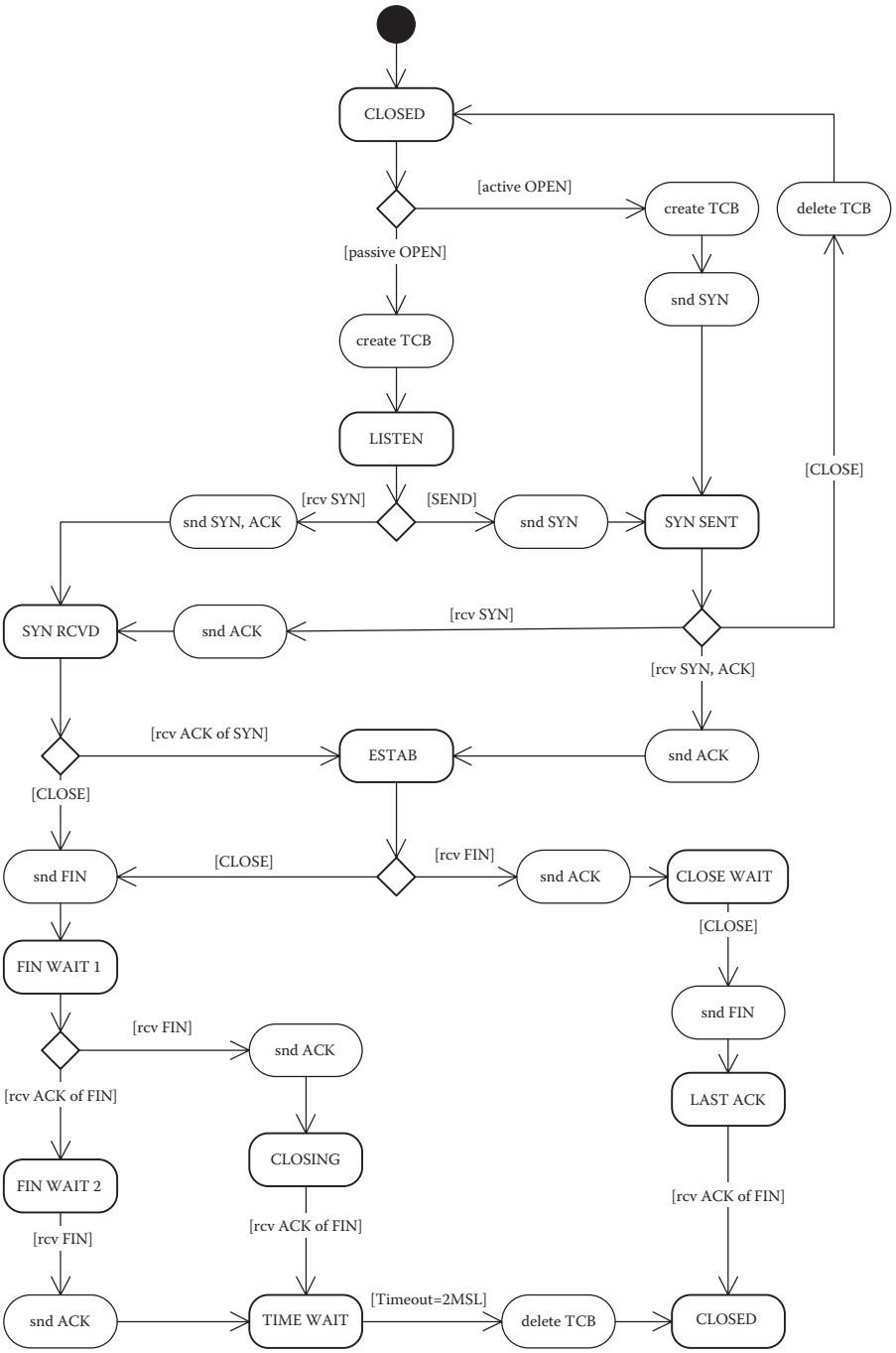


FIGURE 3.20
TCP activity diagram.

abstraction that is the closest to it, and that is the activity state. The model of the TCP protocol shown in Figure 3.20 comprises the following activity states (the names of the states are taken from the RFC 793):

- CLOSED (no connection exists)
- LISTEN (wait for a connection request from any remote TCP and port)
- SYN SENT (wait for a matching connection request after having sent a connection request)
- SYN RCVD (wait for a confirming connection request acknowledgment after having both received and sent a connection request)
- ESTAB (the connection is established, i.e., open)
- FIN WAIT 1 (wait for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request that was previously sent)
- CLOSING (wait for a connection termination request acknowledgment from the remote TCP)
- FIN WAIT 2 (wait for a connection termination request from the remote TCP)
- TIME WAIT (wait for enough time to pass to be sure that the remote TCP has received the acknowledgment of its connection termination request)
- CLOSE WAIT (wait for a connection termination request from the local user)
- LAST ACK (wait for an acknowledgment of the connection termination request previously sent to the remote TCP, which includes an acknowledgment of its connection termination request)

The activity diagram shown in Figure 3.20 is fully compliant with the original TCP standard. Interested readers can refer to IETF RFC 793 for more details.

The last example in this section shows a model of a simplified send e-mail operation. The corresponding activity diagram (Figure 3.21) is a straightforward implementation of a typical SMTP scenario (client side), which has already been introduced in this chapter (Figure 3.12) and in Chapter 2 (Figure 2.12). Although simplified, in the sense that it just follows the successful path of the SMTP scenario, it is a complete specification of a desired behavior because it covers all possibilities in a non-overlapping manner.

Again, like the previous example, the events associated with the reception of the corresponding messages are modeled as guard expressions, while the actions taken by the SMTP client are modeled by the corresponding action states. Additional similarity with the previous example is that the SMTP

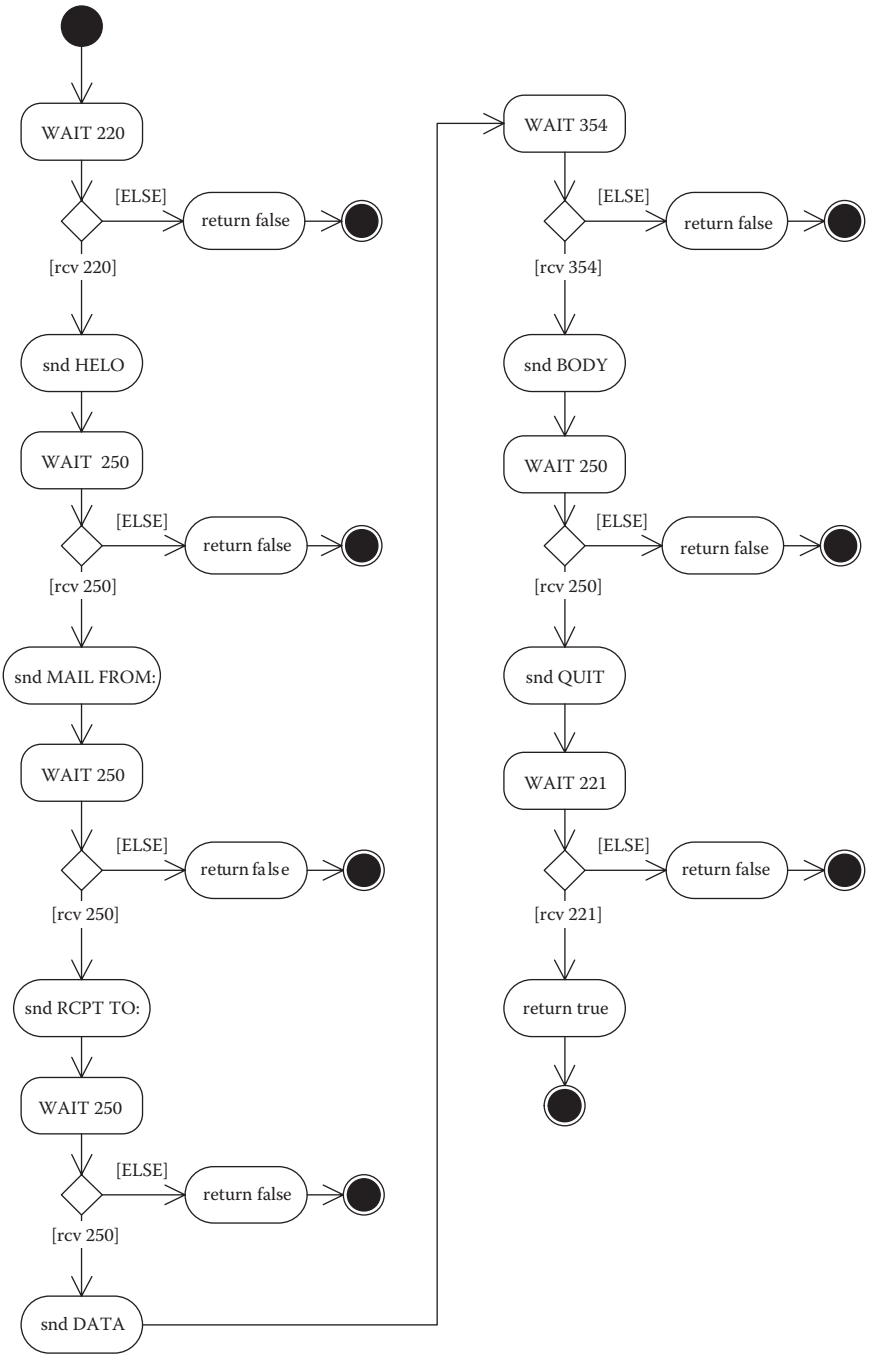


FIGURE 3.21
Simple send e-mail operation activity diagram (SMTP client side).

client, like the TCP protocol, spends most of its time in its stable states, waiting for a message from the SMTP server. If the received message is the one expected, the SMTP client sends the next message, prescribed by the ideal SMTP scenario, and proceeds to the next stable state. If the received message is not the one expected, the SMTP client returns the *value false* and the operation terminates.

The e-mail is successfully sent if all of the prescribed messages between the SMTP client and server are successfully exchanged. In this case, the send e-mail operation returns the *value true* and terminates.

3.5 Statechart Diagrams

In contrast to activity diagrams—which can be used for modeling activities both inside the individual objects and across the workflow of objects—the statechart diagrams are normally used for modeling the lifetime of a single object, typically, an instance of a class or a use case. The activity diagrams emphasize the flow of the action and the activity states, whereas the statecharts emphasize the event-ordered behavior of an object, which is especially suitable for modeling reactive systems.

The common feature of both activity diagrams and statechart diagrams is that they aim at making complete models of behavior, i.e., for use in the design back-end. The driving forces for providing complete behavior specifications are the same, namely, the complete coverage of possibilities without overlaps. The styles differ a bit. By an unwritten rule, the decision symbols are extensively used in activity diagrams and seldom used in statechart diagrams. Therefore, the coverage of possibilities is shown explicitly in activity diagrams and more implicitly in statechart diagrams.

That the activity and statechart diagrams are semantically equivalent is also important to emphasize, i.e., we can use both of them for modeling the same behavior on a comparable level of details. They merely provide two different views of the same behavior. The activity diagrams are better suited for modeling individual operations, whereas the statechart diagrams are better for modeling the behavior of entire stateful objects, especially if the behavior is driven by events (messages).

Statecharts were originally invented for modeling state machines, which makes them a perfect tool for modeling communication protocols because the protocols are essentially state machines. According to the UML terminology, a **state machine** is a sequence of states an object goes through in its lifetime. A **state** is a situation during which an object satisfies a certain condition, performs an activity, or waits for an event. An **event** is an occurrence of a stimulus that triggers the state transition. An **action** is an atomic executable statement (computation). An **activity** is a non-atomic execution composed of actions and

other activities. A **transition** is a relation between the source and the target states (these can be different states or the same state) that specifies the actions to be taken when the given event occurs and the given guard condition is satisfied.

The key abstractions in the context of state machines are the object state and the state transition. We can think of the object state as a period of an object's lifetime (it can be just a moment characterized by a certain condition, a period of a certain activity, or an interval of time in which the object waits for a certain event). Alternately, we can think of the state transition as a rather short interval of object's lifetime, which is related to actions caused by a certain event, and is defined by the following five attributes:

- The source state
- The event trigger
- The guard condition
- The actions
- The target state

A statechart diagram is a special type of graph that comprises a set of vertices that are interconnected by arcs. The basic set of graphical symbols available for rendering statechart diagrams is shown in Figure 3.22. Each symbol has a set of properties that must be set by the designer once they add the symbol to a diagram.

The initial state has three categories of properties. These are the general information, the table of constraints, and the tagged values (documentation notes). The general information is just the name and the type (initial). Each statechart diagram must start with this symbol.

The final state has the same categories of properties as the initial state symbol, with the exception that its type is final. If the lifetime specified by the statechart diagram is infinite, the diagram will not contain this symbol. Alternately, it can contain one or more such symbols.

The state has six categories of properties. These include the general information, the table of entry actions, the table of exit actions, the table of internal transitions, the table of constraints, and the tagged values. The general

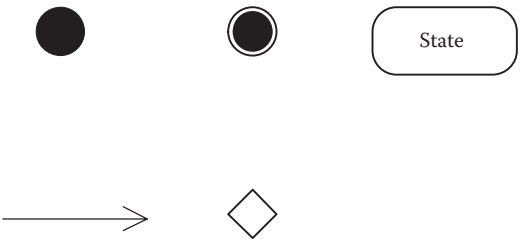


FIGURE 3.22
Basic set of symbols available for rendering statecharts.

information is just the name and the stereotype. Both the table of entry and the table of exit actions store the corresponding action names and their types. The table of internal transitions comprises their properties. Each internal transition is characterized by its name, its stereotype, and the event that triggers the transition.

The following eight common types of actions are used:

- Create an object
- Destroy an object
- Call an operation on another object
- Call an operation on this object (local invocation)
- Send a signal (message) to another (or this) object
- Return a value
- Terminate execution
- Uninterrupted action (other unclassified types of actions)

Four common types of events are also used:

- Signal event: This object has caught (received) the signal (message) that was thrown (sent) by another (or this) object. In UML, we model the signal by the class stereotyped as *<<signal>>*. We can also use a dependency relation, stereotyped as *<<send>>*, between the operation of the class that sends the signal and the class that defines the signal to explicitly show the source of the signal. A signal is an asynchronous event.
- Call event: The object's operation is called by another (or this) object. A call event is a synchronous event. The event name and the parameters are the names and the parameters of the corresponding operations, respectively.
- Change event: The given condition is satisfied. Generally, the condition is related to the state of this object (value of its attributes) or to absolute time. We use the keyword *when* to specify the condition, e.g., *when((time == 17:00)*, or *when(key == pressed)*. A change event is an asynchronous event.
- Time event: The given interval of time has expired. We use the keyword *after* to specify the expression that evaluates to a period of time, e.g., *after(10s)*, or more symbolically *after(T1)*, which means that the timer *T1* has expired. By default, the starting time of such an expression is the time since entering the current state. If we want the starting time to be other than that, we must specify it explicitly. We should note that time events enable implicit timer management, as will be illustrated shortly.

The transition has four categories of properties. These are the general information, the table of actions, the table of constraints, and the tagged values (documentation notes). The general information comprises the name and optionally the corresponding event and the guard expression. The table of actions holds action names and their types. The decision has three categories of properties, namely, the general information (just the name), the table of constraints, and the tagged values (same as the decision in activity diagrams).

Simple examples that illustrate the usage of the basic set of graphical symbols for rendering statechart diagrams seem to be appropriate at this point. The following two examples, shown in Figures 3.23 and 3.24, are semantically equivalent to the simple examples of activity diagrams shown in Figures 3.14 and 3.15, respectively. The activity diagram shown in Figure 3.14 illustrates a sequence of three activity states, namely, *openPort(p)*, *sendData(seg)*, and *closePort(p)*. Figure 3.23 shows three versions of statechart diagrams that model the same behavior. These are the versions A, B, and C.

Version A models the behavior by a sequence of three transient states, namely, *Opening*, *Sending*, and *Closing*. By selecting appropriate names, we can indicate what type of activity is taking place in each of the states. The original activities *openPort(p)*, *sendData(seg)*, and *closePort(p)* are modeled as internal transitions of the states *Opening*, *Sending*, and *Closing*, respectively. We could also use entry or exit actions instead of internal transitions. Alternately, we could model this simple behavior by only one transient state with three internal transitions. Generally, by compressing models we decrease their clarity, and we should seek the compromise appropriate for the project at hand. Of course, defining clarity is tricky because it is essentially a matter of taste.

Version B is the model of the same behavior that employs another way of modeling activities in the statechart diagrams, and that is by actions taken by state transitions. This version of the model comprises three transient states, namely, *Initial*, *Ready*, and *Finished*, which are connected by triggerless transitions. Such transitions take place immediately after their source state is left (finished). The original activities *openPort(p)*, *sendData(seg)*, and *closePort(p)* are modeled here by the actions of the corresponding state transitions.

Finally, version C is the most compressed form of the model with the equivalent semantics. It comprises only one state transition, from the initial to the final state, which conducts a series of actions, namely, *openPort(p)*, *sendData(seg)*, and *closePort(p)*. This extreme shows the power of statechart diagrams. Generally, statecharts are more expressive than activity diagrams when it comes to modeling state machines, therefore we can model the same behavior in less space.

The activity diagram shown in Figure 3.15 is a model of a reliable packet delivery operation, which starts the timer T1, sends a packet, and waits for the answer from the remote site. If the timer T1 expires before the answer is received, the packet is sent again. If the answer is ACK, the operation returns

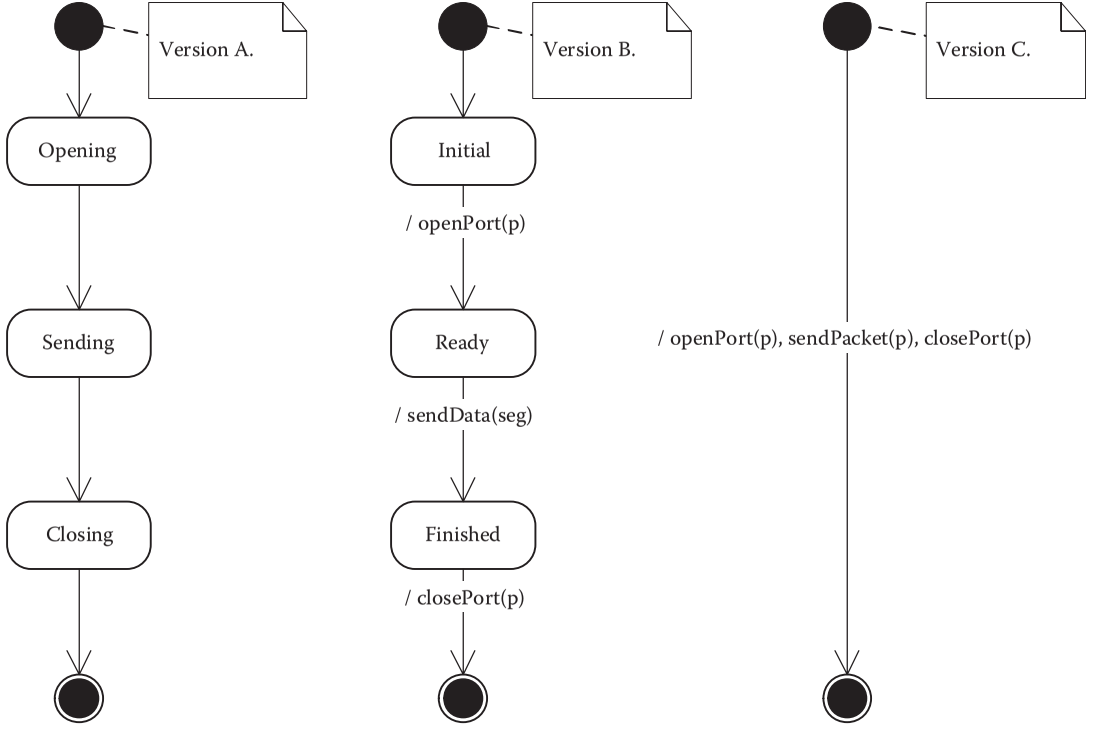


FIGURE 3.23
Example of a simple state machine with a single path of evolution.

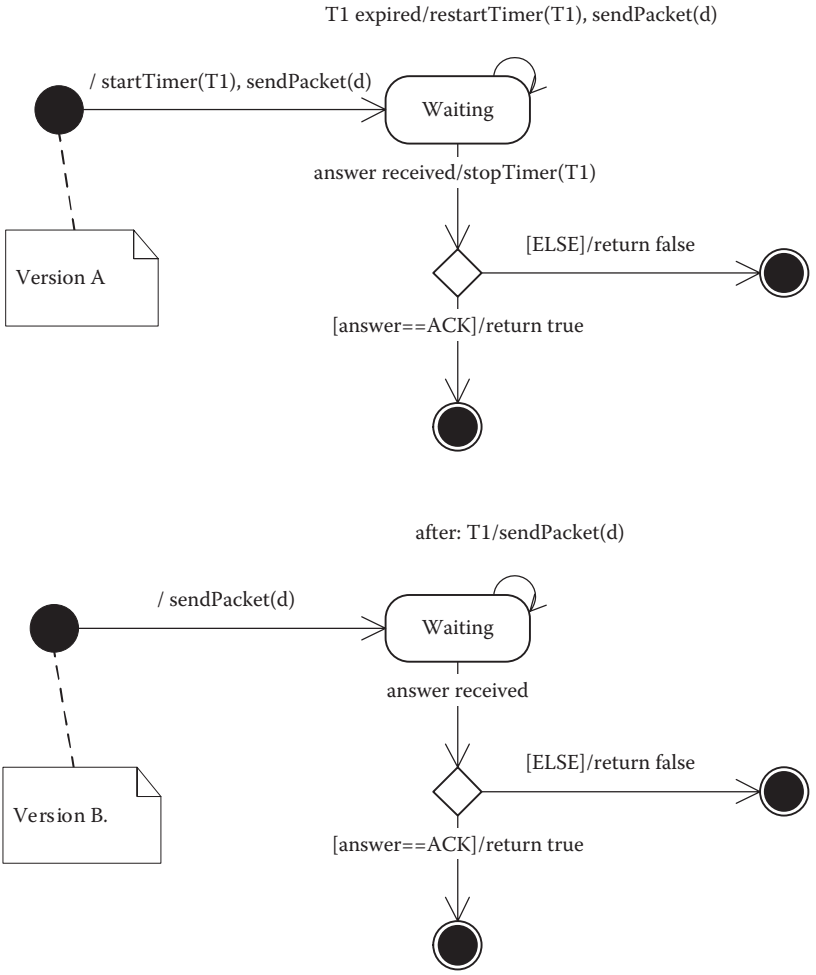


FIGURE 3.24
An example of a simple state machine with alternative paths and loops of evolution.

the *value true*. Otherwise, it returns the *value false*. Figure 3.24 shows two versions of statechart diagrams that are models of the same behavior, namely, versions A and B.

Version A models the given behavior by explicit, rather than implicit, timer management. The triggerless transition from the initial state to the *Waiting* starts the retransmission timer T1 and sends the packet by conducting the actions *startTimer(T1)* and *sendPacket(d)*. The expiration of the timer T1 is modeled here by the signal event *T1 expired*. The corresponding transition restarts the timer T1 and sends the packet again. The reception of the answer from the remote site is modeled by the signal *answer received*. The corresponding transition stops the timer T1 and leads to the decision with

two outgoing transitions. The first is taken if the answer is ACK; otherwise, the second is taken. Those who prefer not to use decision symbols in their statechart diagrams should delete it along with the previous transition, and add the event answer received to both transitions that lead to the final state.

Version B, in contrast to version A, models the given behavior by implicit timer management. Here the triggerless state transition from the initial state to the *Waiting* state just sends the packet by conducting the action *sendPacket(d)*. The existence of the state transition triggered by the time event *after: T1* implicitly implies that the timer T1 has started at the entrance of the *Waiting* state. If the timer T1 expires, the packet is sent again by the action *sendPacket(d)* and the timer T1 is restarted at the new entrance to the *Waiting* state. The event *answer received* occurs when this object receives the answer from the remote side. This event triggers the transition that leads to the decision and, later, to the final state. The timer T1 is implicitly stopped at the exit from the *Waiting* state. The result is a more compressed form of a model with more implicit details, which may not be seen at first glance. We can use either one of these two styles, but we should be consistent and stick to one on a certain project.

Now that we have covered the basics of statechart diagrams, we proceed to their more advanced abstractions. First, besides entry and exit actions and internal transitions, a state can perform an ongoing activity that we can specify by using the keyword *do*. Most of the states are stable states, which means that the object is blocked while waiting for an event. Some of the states are transient, which means that they perform certain computations and then finish. Sometimes we need to also model active states, which perform some activities while simultaneously waiting for an event to occur; we do these by using the keyword *do*. Generally, the special *do* transition can name another state machine or a sequence of actions.

Deferred events are the next important abstraction in the context of states. Until now, we were not interested in the events that occur during the state that does not react to them. What happens to these events? They are simply lost. If we want to save them so that they can be processed later in some other states, we must specify that they are to be deferred by using the special action named *defer*. Each event that is associated with this special action will be saved for further processing by the states that explicitly name that event in one of their transitions.

We have already shown how to manage complexity by using hierarchical organization. Statechart diagrams allow us to use that powerful concept in the context of states. Until now, we have dealt with simple states. Actually, a state in UML can also be a composite state, which means that it can comprise simple states and other composite states. This nesting of states can go to an unlimited depth, at least in theory.

A composite state can contain either sequential or concurrent substates. The sequential substates are disjoint, i.e., an object can be in only one of them at a certain point in time. The concurrent substates are orthogonal,

which means that an object at a certain point in time is in all of the concurrent substates that are active at that point. We can think of a concurrent state as one aspect (orthogonal axis) of the object's lifetime.

The state transitions until now were transitions between simple states. After the introduction of composite states, the situation becomes more complex in this respect. Besides the transitions between simple states, there exist the transitions between simple states and composite states, as well as the transitions from substates to external states. The transitions from external states to substates of a composite state are not allowed. This asymmetrical relation raises the following question: What happens to the flow of states inside a composite state if a transition from that composite state to another state is triggered?

The answer is that the information about the point of interruption inside the composite state is lost by default. This means that the processing will be restarted from the very beginning when that composite state is reentered once again later. This means that the composite state operates without context saving, which is referred to as a **history** in the UML.

If we want the composite state to operate with the history—which means it is able to restart from the point of interruption at its reentrance—we can use the special history state. The history state is a special type of an initial state that is the target for the transitions from the external states. Once activated, it restarts the operation at the point of interruption. The following two types of history states are used:

- The shallow history state (marked with the symbol H)
- The deep history state (marked with the symbol H*)

The shallow history state ensures context-saving only on the first level of nesting of composite states. Alternately, the deep history state provides context-saving on the innermost state at any depth. If there are more nesting levels, the shallow history remembers the outermost nested state and the deep history remembers the innermost nested state.

Like activity diagrams, statechart diagrams also support modeling concurrency. We model concurrent activities in statechart diagrams by using concurrent sequences of substates inside a certain composite state. Typically, each such sequence begins with the initial state and ends with the final state. The transition from the external state to this composite state forks to concurrent substates, which at the end joins in the transition from this composite state to the external state. The usage of concurrent substates is advisable only if the behavior of one of these concurrent flows is affected by the state of another. Alternately, if the behavior of the concurrent flows is driven by the signals (messages) they exchange, partitioning the object into more active objects is preferable.

The set of additional symbols that are available for rendering statechart diagrams is shown in Figure 3.25. These are the composite state, the shallow history state, the deep history state, the fork or join synchronization point, the note, the constraint note, the constraint, and the OR constraint. These symbols, like others, have their properties. The composite state has the same categories of properties as a simple state, plus two additional indicators (*Concurrent* and *Region*) which determine whether the composite state is concurrent or not and if it is a region or not. Both shallow and deep history states have the same three categories of properties. These are the name, the table of constraints, and the tagged values. The rest of the symbols have already been introduced.

Figure 3.26 shows the simple example of a statechart diagram that uses the shallow history state. Imagine a simple state machine that starts from the state *Idle*. The event *sendCharacter(ch)* triggers its transition to the composite state *Sending Segment*, which starts with the shallow history state to ensure context saving. Because this state comprises only simple states, the application of the deep history state, instead of the shallow history state, would have the same effect because only one level of nesting of composite states is found.

The state machine remains in the substate *Buffering* while it is filling the corresponding buffer with new incoming characters. This status means that the state machine will wait for the additional event *sendCharacter(ch)* until the buffer becomes full, when the state machine will proceed to the state *Sending*. After it sends all the characters from the buffer, the state machine leaves the compound state *Sending Segment* and triggerlessly transits to the state *Idle*.

If the event *break* occurs while the state machine is in the compound state *Sending Segment*, its context will be saved and the state machine will leave it and move to the state *Break*. It will remain in this state until the event *continue*

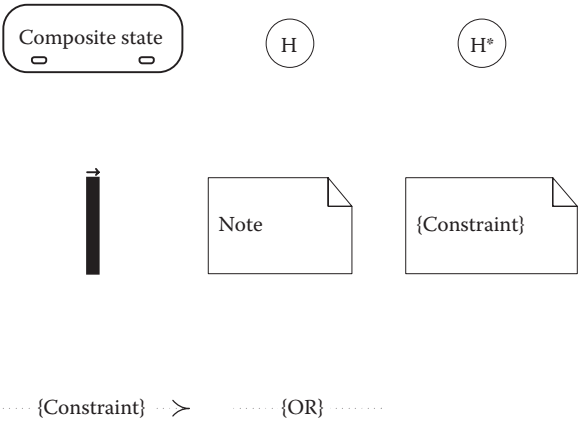


FIGURE 3.25
Additional graphical symbols available for rendering statecharts.

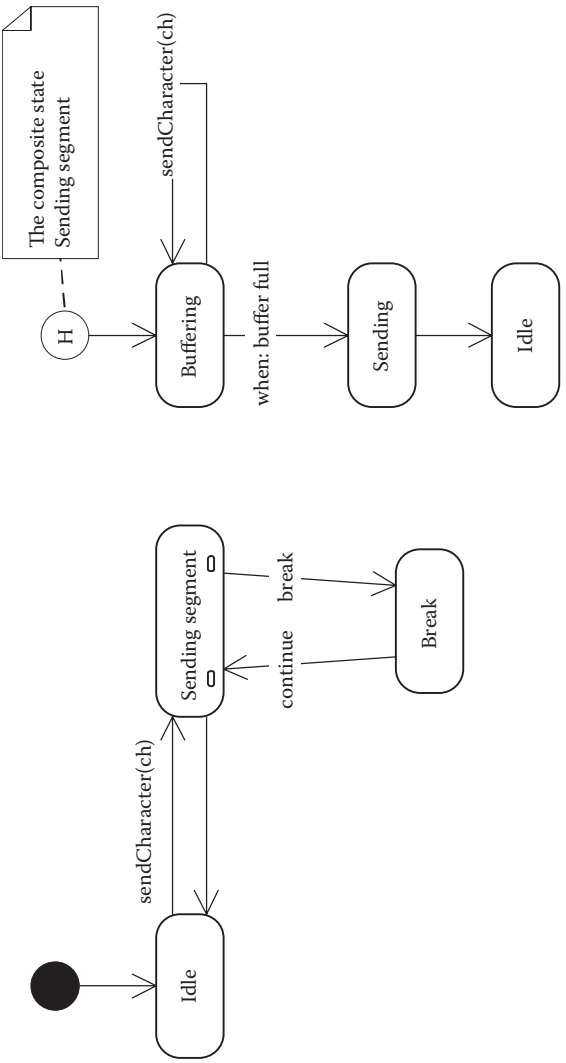


FIGURE 3.26
Example of a composite state that uses the shallow history state.

occurs. Then the state machine will reenter the compound state *Sending Segment*, the context will be restored, and the state machine will resume the processing from the point of interruption.

The example in Figure 3.27 shows simplified DNS client and server statechart diagrams. Both of them have just a single state. Being simple enough, these diagrams make very clear how statechart diagrams are used to make complete designs of communication protocols. Typically, a job performed by the communication protocol is to receive a message, process it, and send one or more messages as the result of this processing. Both DNS client and server go along this simple scheme.

The DNS client starts from the initial state by receiving a call to map the given domain name into the corresponding IP address. This action is modeled by the call event *map(d)* in Figure 3.27. This event triggers the transition of the DNS client from the initial state to the state *Wait DNS Response*. During the course of this transition, the DNS client sends the signal (message) *DNSrequest(d)*, which causes the signal event *receive DNSrequest(d)* at the DNS server side.

The DNS client is simply blocked in the state *Wait DNS Response* while waiting for the signal *DNSresponse(d,ip)*. The signal event *receive DNSresponse(d,ip)* triggers the DNS client transition to its final state. During this transition, the DNS client extracts the IP address from the received signal and returns it as its return value. This is modeled by the return action *return(ip)*.

The DNS server starts with the triggerless transition from its initial state to the state *Wait DNS Request*, where it is blocked while waiting for the signal *DNSrequest(d)*. The signal event *receive DNSrequest(d)* causes the DNS server

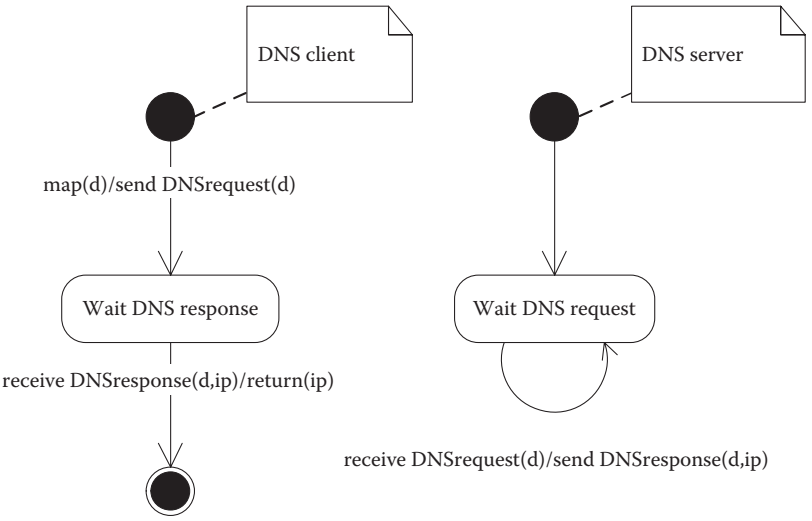


FIGURE 3.27
DNS client and server statechart diagrams.

to map the given domain name to the corresponding IP address, to create the signal (message) *DNSresponse(d,ip)*, and to send it to the DNS client. The DNS server performs all these actions during the transition to the same state, i.e., *Wait DNS Request*. This ensures that after servicing the current request, the DNS server remains available for servicing the next DNS request.

The example in Figure 3.28 shows the statechart diagram for one real protocol, namely TCP. It starts with the triggerless transition from the initial state to the state *CLOSED*, in which it awaits one of the two possible call events. The call event *passive OPEN* causes TCP to create TCB (modeled with the action *create TCB*) and to move to the state *LISTEN*. Alternately, the call

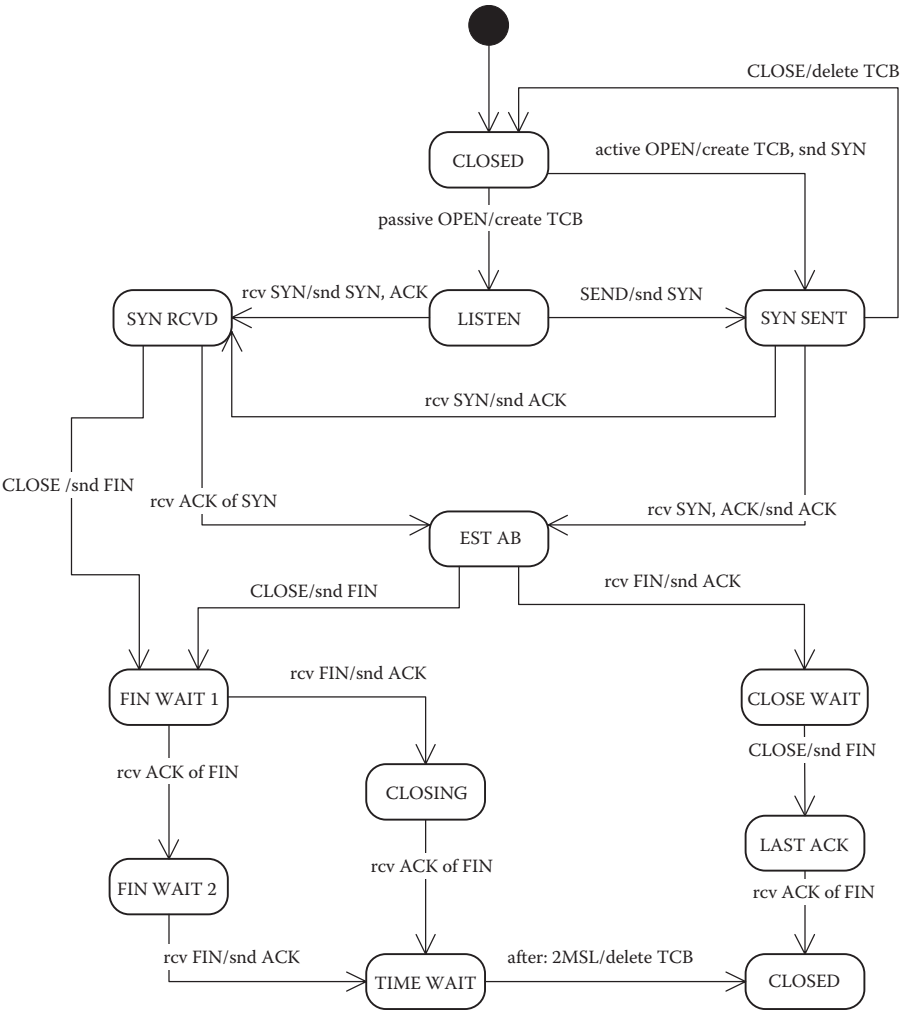


FIGURE 3.28
TCP statechart diagram.

event *active OPEN* causes TCP to additionally send the signal *SYN* (TCP segment with the bit SYN set in the header) to the remote TCP entity. This is modeled with the actions *create TCB* and *snd SYN*.

TCP is blocked in the state *LISTEN* while waiting for one of the two possible events. The signal event *rcv SYN* triggers it to send the signal *SYN, ACK* (TCP segment with both bits SYN and ACK set) to the remote TCP entity and to move to the state *SYN RCVD*. The call signal *SEND* causes TCP to send the signal *SYN* to the remote TCP entity, and to move to the state *SYN SENT*.

While blocked in the state *SYN SENT*, TCP can be triggered by one of three possible events. If the call event *CLOSE* occurs, TCP deletes TCB (modeled with the action *delete TCB*) and returns to the initial state. If the signal event *rcv SYN* occurs, TCP sends the signal *ACK* and moves to the state *SYN RCVD*. If the signal event *rcv SYN, ACK* occurs, TCP sends the signal *ACK* to the remote TCP entity and moves to the state *ESTAB*.

After reaching the state *SYN RCVD*, TCP can react to one of the two possible events. If the call event *CLOSE* occurs, TCP sends the signal *FIN* to the remote TCP entity and moves to the state *FIN WAIT 1*. If the signal event *rcv ACK of SYN*, occurs, TCP moves to the state *ESTAB*.

Two events are recognizable in the state *ESTAB*. If the call event *CLOSE* occurs, TCP sends the signal *FIN* to the remote TCP entity and moves to the state *FIN WAIT 1*. If the signal event *rcv FIN* occurs, TCP sends the signal *ACK* and moves to the state *CLOSE WAIT*.

In the state *FIN WAIT 1*, TCP may receive either *FIN* or *ACK of FIN* signals. In the former case, it sends the signal *ACK* and moves to the state *CLOSING*, whereas in the latter case it just moves to the state *FIN WAIT 2*, where it waits for the signal *FIN* to send the signal *ACK* and move to the state *TIME WAIT*. On the alternative path, TCP moves from the state *CLOSING* to the state *TIME WAIT* after it receives the signal *ACK of FIN*.

Upon the entrance to the state *TIME WAIT*, a timer with the period 2MSL is started. When this period expires, TCP deletes TCB and moves back to its initial state *CLOSED*. After reaching the state *CLOSE WAIT*, TCP waits for the call event *CLOSE* to send the signal *FIN* and move to the state *LAST ACK*, and from there to the initial state *CLOSED* after it receives the signal *ACK of FIN*.

The example in Figure 3.29 shows the statechart diagram of a simple send e-mail operation (SMTP client side). It starts with the triggerless transition from its initial state to the state *WAIT 220*, where it waits for the signal (message) *220* from the SMTP server. When the signal event *rcv 220* occurs, the SMTP client sends the signal *HELO* to the SMTP server and moves to the state *WAIT 250 1*. After receiving the signal *250*, the SMTP client sends the message *MAIL FROM:* to the SMTP server and moves to the state *WAIT 250 2*.

Next, two signals of *250* in succession cause the SMTP client first to send the signal *RCPT TO:*, then to send the signal *DATA* to the SMTP server, and finally to reach the state *WAIT 354*. Upon reception of the signal *354*, the SMTP client sends the body of the e-mail message and moves to the state *WAIT 250 4*. After receiving the signal *250*, it sends the signal *QUIT* to the

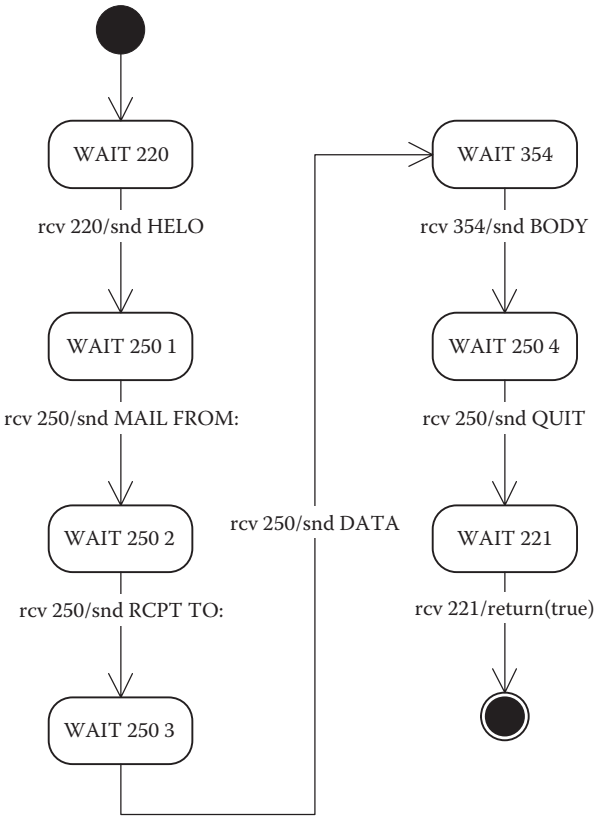


FIGURE 3.29
Simple send e-mail operation statechart diagram (SMTP client side).

SMTP server, and finally, after receiving the signal *250* again, it returns the value *true* and moves to its final state.

The main problem in this oversimplified version of the SMTP client is that it can block indefinitely while waiting for a signal from the SMTP server. The first thing that would be added in a more realistic design is a time limit on waiting for signals, which would be modeled with timers (keyword *after*:). The reaction to the expiration of a timer could be as simple as returning the value *false* and moving to the final state, or it can include some type of a recovery mechanism.

3.6 Deployment Diagrams

Deployment diagrams are used to model the deployment of the components, the component instances, objects, and packages on nodes and node

instances. A **component** is a part of the system that implements a set of interfaces. It typically models a physical package of logical elements, such as classes, interfaces, and collaborations. The common forms of packages are the following:

- Executables
- Libraries
- Tables
- Files
- Documents

A **node** is a physical element that models a computational platform, which comprises a set of resources, such as memory banks, buses, I/O channels, controllers, processors, and so on. The examples of nodes are the following:

- Personal computers
- Mainframes
- Embedded controllers
- Mobile or cellular phones
- Network routers

We use deployment diagrams in the design phase of communication protocol engineering for the following two main purposes:

- To identify network nodes and configurations
- To identify design subsystems and interfaces

The software architecture is closely related to the structure of the physical network. Sometimes the latter can be fixed and, in such a case, it governs the distribution of functionality across the network nodes as well as the selection of active classes. Alternately, both software architecture and network structure can be subjects of design and, in that case, some particular network structures can yield more appropriate software architecture and system solutions.

While trying to identify network nodes and configurations, we typically render network nodes as cubes, interconnect them with association relations, and think how to deploy individual components on these nodes. We show the deployment in the deployment diagrams by adding the component symbols (rectangles with tabs) and by connecting the related nodes and components with the dependency relations. Another way to do this is to adorn the node instances by the names of the components that are deployed on them.

Similarly, while trying to identify the subsystems and interfaces, we typically render the packages with their corresponding interfaces. We try to

organize them into hierarchical layers (e.g., application-specific, application-general, middleware, and system-software). Finally, we show which interfaces (services) are provided by which packages or components and also which packages or components are users of the services provided through those interfaces.

Deployment diagrams are a special type of graph that comprise the set of vertices that are interconnected with the corresponding arcs. Figure 3.30 shows the basic set of graphical symbols available for rendering deployment diagrams. These are the node, the node instance, the component, the component instance, the object, the package, the interface, the association relation, the aggregation relation, the dependency relation, the note, the constraint

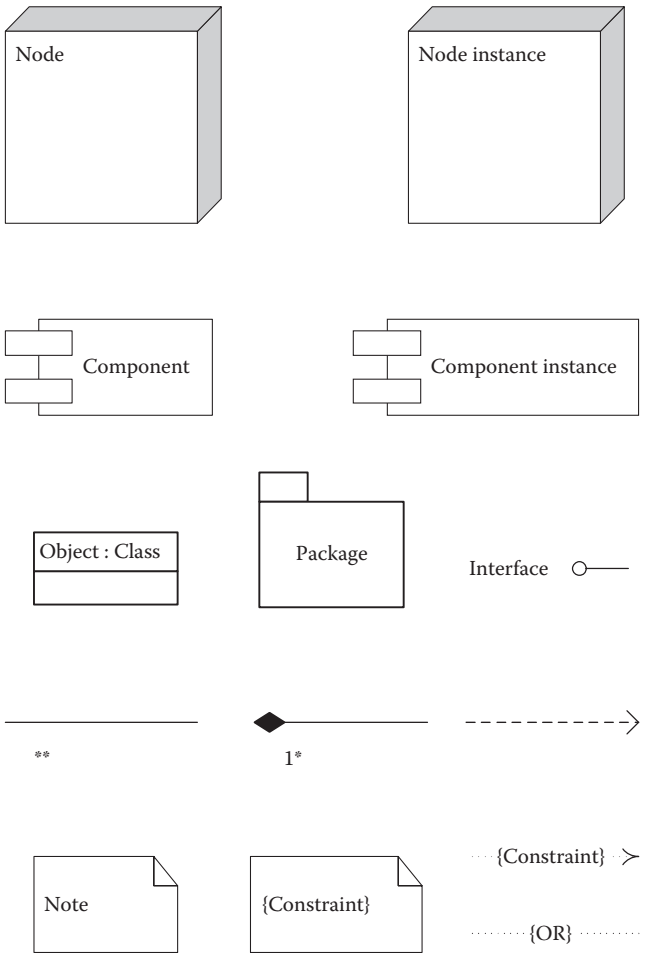


FIGURE 3.30
Basic set of symbols available for deployment diagrams.

note, the two-element constraint, and the OR constraint. Each symbol has a set of properties, which must be set by the designer once they add the symbol to the diagram. The new symbols are the symbols representing the nodes, the components, and their instances. The rest of the symbols are already introduced in the previous sections about class and object diagrams (called together a static structure).

The node has six categories of properties. These are the general information, the table of attributes, the table of operations, the list of components, the table of constraints, and the tagged values. The general information includes the name, the full path, the stereotype, the visibility, and the indicators *Root*, *Leaf*, and *Abstract*. The list of the components comprises the names of the components that are deployed by this node.

The component has seven categories of properties, including the general information, the table of attributes, the table of operations, the list of nodes, the list of classes, the table of constraints, and the tagged values. The general information comprises the name, the full path, the stereotype, the visibility, and the indicators *Root*, *Leaf*, and *Abstract*. The list of nodes holds the names of the nodes that deploy this component. The list of classes stores the names of the classes that are implemented in this component.

The node instance has four categories of properties: These are the general information, the table of attribute values, the table of constraints, and the tagged values (documentation and persistent). The general information comprises the node instance name and the node name. The table of attribute values stores the name, the stereotype, the type, and the value for each attribute. The component instance has the same categories of properties as the node instance, with the exception that its general information differs and it comprises the name of the component instance and the component name.

The deployment diagram in Figure 3.31 shows an example of a network configuration comprised of three personal computers that are connected to the Internet. A personal computer is modeled as the node *PC*. Individual PCs are modeled as node instances, namely *Machine1*, *Machine2*, and *Machine3*. The Internet is modeled as the node instance, named *Network*, of the node type named *Internet*. The real links that connect PCs to the Internet are modeled with the association relations between the node instances *Machine1*, *Machine2*, and *Machine3*, and the node instance *Internet*. The one-to-one nature of these links is modeled by setting the multiplicities on both sides of the associations to 1.

This diagram is what the physical infrastructure of this example looks like. The software components are deployed as follows: The e-mail client executable is deployed to the first PC, the DNS server executable is deployed to the second PC, and the SMTP server is deployed to the third PC. We model the e-mail client executable with the component *EMailClient*, which is stereotyped as the *<<executable>>*, and its particular instance is deployed to the first PC with the component instance *client.exe*. Similarly, the DNS server

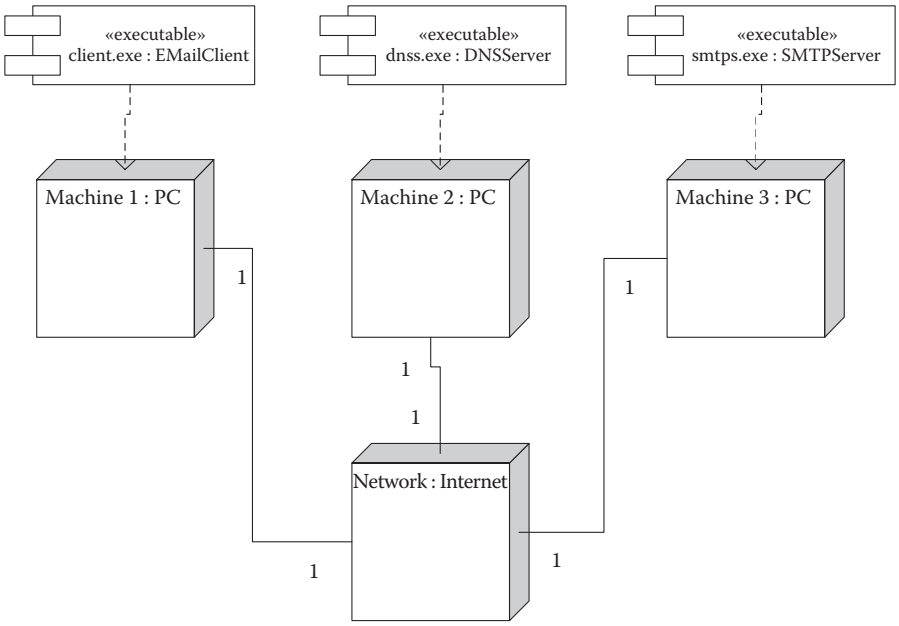


FIGURE 3.31
Example of a network configuration.

executable is modeled with the component *DNSServer* and its particular instance is deployed to the second PC with the component instance *dnss.exe*. Finally, the SMTP server is modeled with the component *SMTPServer* and its particular instance is deployed to the third PC with the component instance *smtps.exe*.

The deployment diagram in Figure 3.32 shows the example of subsystems and interfaces. While thinking about the system shown in the previous example (Figure 3.31), we can identify three application layer packages, two system-software layer packages, and three interfaces. The application layer packages are the packages *EMailClient*, *SMTPServer*, and *DNSServer*, whereas the system-software packages are the packages *TCP/IP* and *OS*.

The package *TCP/IP* provides two service types through the interface *TCPport* and *IPint*, respectively. The services provided through the former interface are used by the package *EMailClient* and *SMTPServer*, whereas the services provided through the latter interface are used by the package *EMailClient* and *DNSServer*. Similarly, the package *OS* provides services through its interface *OSapi*. These services are used by the package *TCP/IP*.

Interested readers can find more information about the UML diagrams in the original books by Booch, Rumbaugh, and Jacobson (Booch et al. 1998). This section concludes the part of this chapter based on UML. The second part of the chapter will be based on domain-specific languages.

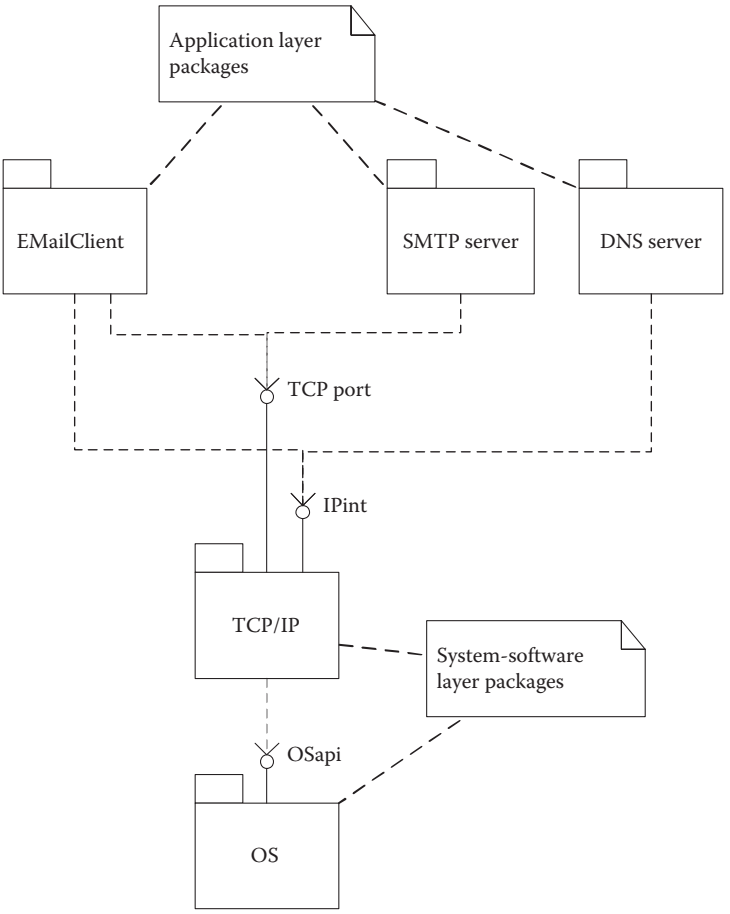


FIGURE 3.32
Example of subsystems and interfaces.

3.7 Specification and Description Language

Software for real communication systems and devices (concentrators, packet switches, gateways, routers, and so on) is very complex and, therefore, hard to understand. Proving that this type of software is correct is very difficult; thus, special attention is paid to its design. Software of this type can be modeled in the form of an individual or a group of finite state machines. Japanese designers were the first to apply this method of specification and description of communication protocols in the 1970s. Not long after its initiation, the CCITT (predecessor of ITU-T) has standardized it in the form of the so-called Specification and Description Language (SDL).

SDL creators have been facing the following dilemma. Traditionally, a finite state machine (FSM) has been modeled by a state transition graph. Typically, a state transition graph is graphically illustrated by circular symbols representing states and arrows representing state transitions. State labels are state names, whereas state transition labels indicate FSM input that causes the corresponding state transition and FSM output produced by the same transition. An advantage of this type of FSM representation is that all the stable FSM states are clearly indicated and can be easily noticed. Alternately, a disadvantage of this type of FSM representation is that message-processing procedures are not defined formally. Informally written state transition labels, placed close to the corresponding arrows, indicate only the FSM input causing the transition and the output that the FSM must produce. This information is far from being sufficient for writing the software that implements the given FSM—it only provides some hints to programmers.

Another approach would be to use a flowchart, a traditional way of specifying data-processing algorithms. An advantage of this type of FSM representation is that message-processing procedures are clearly and precisely defined. A disadvantage is that stable FSM states are not clearly indicated, therefore they can hardly be noticed. The FSM states can be marked as certain points in a flowchart by using informal annotations, and that is simply not comprehensible enough.

The creators of the SDL language have found a solution to this dilemma by combining the abovementioned approaches, namely, the state transition graph-based approach and the flowchart approach. This combination has been cleverly made by simple extension of the set of graphical symbols available for drawing flowcharts. The key new graphical symbols introduced are the symbols corresponding to an FSM stable state and the symbols that represent FSM inputs and outputs (input and output messages). We will fully describe all the SDL graphical symbols later in this chapter.

The protocol designer uses SDL language to specify and describe the corresponding automata instance by listing all its states and state transitions. Although the number of states can be very large, this task is simplified by the fact that in a given state, only a limited number of events can occur. This means that the automata instance can evolve from a given state only for a limited number of new states. For example, consider a telephone call automata instance waiting for the first digit to be dialed (the automata instance enters this state immediately after the user has initiated an outgoing call, i.e., after the so-called "hook-off" event). The telephone call automata instance cannot evolve from this state to any other arbitrary state. More precisely, in this state only the following three events are possible:

- The user ends the call (hook-on event), which causes the automata instance to evolve to its initial idle state.

- The user dials a digit (digit event). This event triggers the state transition from the current state to the state of waiting for the second digit.
- The user does nothing during a certain interval of time. This will cause the expiration of the corresponding timer and a state transition to the state in which the telephone line is blocked.

Communication protocol is by nature a reactive system. Normally, it is blocked in its current state while waiting for one of a few recognizable events to occur. Statistically, it is inactive most of the time. A recognizable event triggers the corresponding state transition to a new state, where the protocol is again blocked while waiting for further events. The state transitions comprise a finite number of primitive operations that are statistically rather short.

An important characteristic of program implementations of the protocols is that they are not trying to monopolize the CPU. This implies that the execution of this type of a program should be organized as a **process with stable states**. In contrast to the conventional time-slicing system, where the task switching is driven by timer interrupts, the switching of processes with stable states is performed at the moment at which the running process reaches its new stable state. Whereas conventional tasks can be interrupted in an arbitrary point of time (determined by the asynchronous occurrence of a timer interrupt signal), a process with stable states is normally not subject to preemption because, unlike conventional tasks, they are not monopolizing the processor. Of course, a process with stable states can be interruptible so that the whole system can react to the urgent events handled by the higher priority tasks.

Enumeration of the possible states and state transitions, as described above, is a logical process that seems to be straightforward for the experts. However, graphical language, such as SDL, is needed to make it possible for design engineers to easily make complete formal specifications of the protocols. The main advantages of graphically oriented languages are as follows:

- Graphical language is easy to read and, because of that, it is easy to check specification completeness and correctness.
- The specification can be easily extended.
- The specification can be directly implemented in software. This means that if the specification is correct, a high probability exists that the software implementation is also correct.

According to ITU-T, the complete software (system) is decomposed into a set of **functional blocks**. Each functional block consists of a set of processes and each process comprises a number of **tasks** (Figure 3.33).

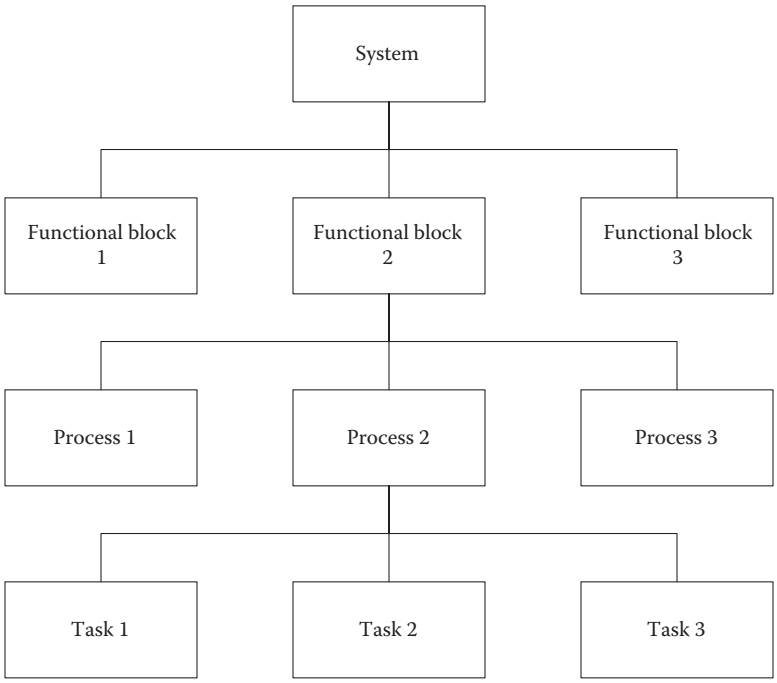


FIGURE 3.33
Structure of the communication software according to ITU-T.

A **process** is essentially an execution of a logical function, which consists of a series of operations applied to message information elements (referred to as tasks) in discrete points of time. Either it is in some of its stable states or it makes its transition from the current to the next state. (In Chapter 4, we refer to the state transition as **unstable states**).

A **signal** is defined as a data stream that delivers information to the receiving process. A data stream among the processes inside the same functional block represents the **internal signal**, whereas a data stream between the processes that are parts of different functional blocks represents the **external signal** to the receiving process. Therefore, from the receiving process point of view, the signal can be classified as internal or external, depending on whether it originates from the same or from a different functional block.

Today, SDL is a standard design language that can be used to specify and describe any system implemented in hardware or software, particularly real-time systems. In this book, we are especially interested in one type of the real-time systems—communications systems.

The basic set of SDL rules is given in ITU-T recommendation Z.100e. Additional explanations are given in a series of subsequent ITU-T recommendations, namely Z.100d1e, Z.100nce, Z.100nfe, Z.100p1e, and Z.100s1e. The main characteristics of the SDL language are as follows:

- It is easy to learn.
- It is easy to extend the specification in case of the new requirements.
- In principle, it can support various methodologies for making the system specifications.

Two forms of SDL language exist, graphical (SDL-GR) and program (SDL-PR). The graphical form has been widely accepted for two reasons. First, it is closer to human understanding because it is easier to understand and follow. Second, in principle, it does not require the support by special, and frequently very expensive, software tools. Of course, a piece of paper and a pencil is hardly sufficient for a professional work. At least a modern graphical editor that supports the SDL set of graphical symbols is needed to enable the making of decent specifications. In this book, we use Microsoft Visio® for that purpose.

The second SDL form, SDL-PR, is practically a higher-level programming language of textual type (similar to C/C++ and Java programming languages). Clearly, this programming language is less synoptic and is harder to follow than the graphical form. It is intended to be used mainly by the accompanying software tools, such as Telelogic® Software Development Tools (SDTs). The goal of using such software tools is not just to make isolated specification and description documents, but rather to make electronic specifications, essentially models of protocols. The software tools can then be used to interpret the models and generate the corresponding program code.

In addition to the tools provided by Telelogic®, other tools exist based on this philosophy that is, as already mentioned, referred to as model integrated computing (MIC). One of them is also already mentioned, GME.

The main SDL applications are the following:

- Call processing in switching systems
- Error supervision and management in telecommunication systems
- Supervision, control, and data acquisition systems
- Telecommunication services
- Data transfer protocols
- Protocols in computer communications

The SDL language basics are as follows: SDL is based on a set of special symbols and the rules for their application. The graphical form (SDL-GR) is based on special graphical symbols whereas the program form (SDL-PR) is based on a set of special keywords. Both SDL forms use the same set of keywords specialized for data representation.

Later, we assume that a system consists of a number of protocols. Also, we refer to a set of hierarchically organized protocols as a **family of protocols** or a **protocol stack**. Typically, each protocol that is a part of the family performs

its well-defined task. The family of protocols conducts rather complex tasks by cooperation of its members.

A system is described as a set of interconnected functional blocks. **Channels** are defined as communication links that are used for the interblock communication and for the communication between the blocks and the environment. Each block comprises a number of processes that communicate by exchanging signals. A channel is typically implemented as a FIFO (First-In-First-Out) queue that stores the signals (i.e., messages) to be transferred through the channel. A process is defined as a finite state machine (automata instance) that is described by the given set of states and state transitions.

The next simple example illustrates the notions and terms introduced above. Both graphical and program SDL forms are presented. The only goal of presenting the program form is to provide the intuition for the reader that will help them understand the main differences between the graphical and program forms of the SDL language. The aim of this book is not to fully cover the program form of the SDL language.

The example is a simple game called *Daemongame*. The core of the game is a simple FSM that has only two states, *even* and *odd*. Timing is controlled with a single timer. The expiration of the timer (this event is labeled *none*) causes the FSM to switch from an *even* state to an *odd* state. The player presses a button when they wish (this event is labeled *Probe*), i.e., at arbitrary points of time. If the FSM is in an *even* state, the player gets one negative point (*Lose*). If the FSM is in an *odd* state, the player gets one positive point (*Win*). If the player scores more *Win* than *Lose* points, they win the game.

The first step in describing this simple system is to define input and output signals. Input signals are as follows:

- *Newgame*: The player wants to start the game.
- *Probe*: The player has pressed a button.
- *Result*: The player wants to see the current score.
- *Endgame*: The player wants to quit the game.

Output signals are the following:

- *Gameid*: current game identification
- *Win*: positive point
- *Lose*: negative point
- *Score*: total amount of points (number of *Win* points minus number of *Lose* points)

The specification of the game *Daemongame* in the graphical form of SDL is shown in Figure 3.34. It contains a single functional block labeled *Game*.

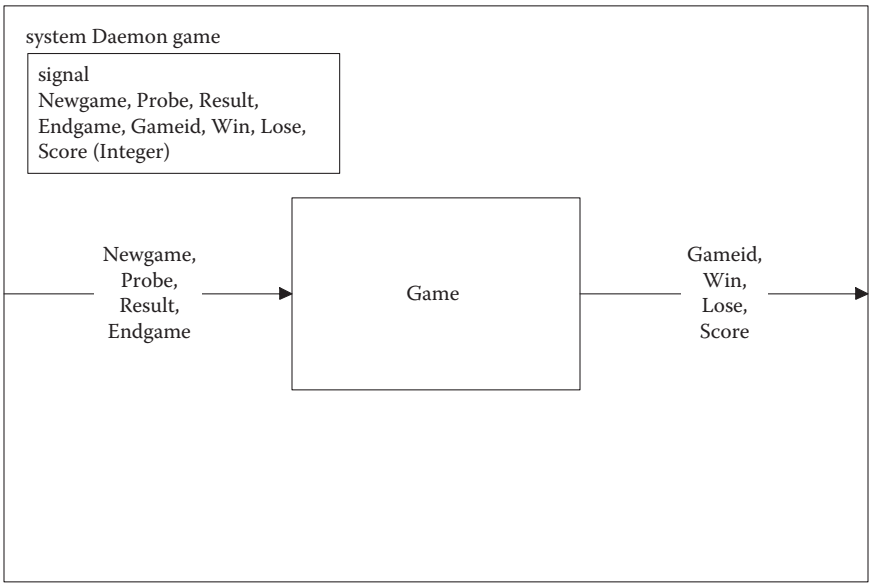


FIGURE 3.34
Structure of the system *Daemongame*.

Input signals are *Newgame, Probe, Result,* and *Endgame*. Output signals are *Gameid, Win, Lose,* and *Score*. Signal declarations are shown in the upper left corner of the figure.

The *Daemongame* system specification in the program form of SDL is as follows:

```
system Daemongame
  signal Newgame, Probe, Result, Endgame, Gameid, Win, Lose, Score(Integer);
  channel Gameserver.in
    from env to Game
    with Newgame, Probe, Result, Endgame;
  endchannel Gameserver.in;
  channel Gameserver.out
    from Game to env
    with Gameid, Win, Lose, Score;
  endchannel Gameserver.out;
block Game referenced;
endsystem Daemongame;
```

Generally, any system SDL program specification starts with the keyword *system* and ends with the keyword *endsystem*. This particular program defines all the required signals (*Newgame, Probe, Result, Endgame, Gameid, Win, Lose,* and *Score*), the input channel *Gameserver.in*, and the output channel *Gameserver.out*.

In contrast with the graphical form, which is easy to understand, the program form represents a lower-level specification, closer to the machine and with more details. For example, in the graphical form a channel is simply represented by an arrow pointing to or from the functional block. The channel declaration in the program form is much more detailed: It comprises the channel name (e.g., *Gameserver.in*), its direction (e.g., from the environment toward the functional block *Game*), and a list of signals that must be transferred over the channel (e.g., *Newgame*, *Probe*, *Result*, and *Endgame*).

The next lower hierarchical level of detail describes a single functional block of this simple system, namely, the block *Game*. Its specification is given in both forms of SDL. The graphical form of the specification is given in Figure 3.35. The program form of the specification is given immediately after a short explanation of Figure 3.35.

Figure 3.35 shows that the block *Game* consists of two processes, namely *Monitor* and *Game*. The processes are connected to the environment and to each other by signaling paths. It also shows that the input channel *Gameserver.in* consists of two signaling paths, the signaling path R1 (which is used to carry *Newgame* signal) and the signaling path R2 (which is used to carry the signals *Probe*, *Result*, and *Endgame*). The output channel *Gameserver.out* comprises the single signaling path R3. A single internal signaling path exists inside the block *Game*, the path R4, which is used to carry the internal signal

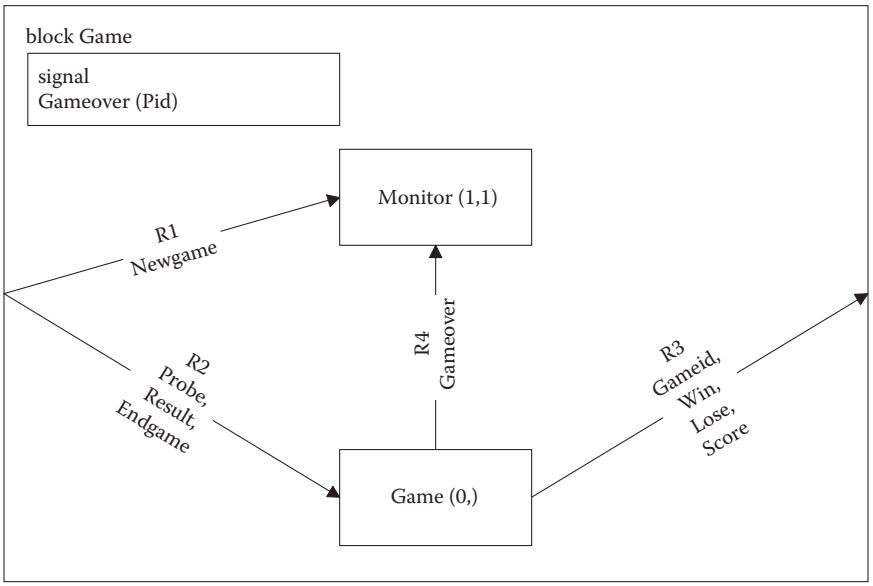


FIGURE 3.35
Structure of the functional block *Game*.

Gameover from the process *Game* to the process *Monitor*. This new signal is declared in the upper left corner of the graphical specification.

The specification of the block *Game* in SDL-PR is as follows:

```
block Game;
  signal Gameover(Pid);
  connect Gameserver.in and R1, R2;
  connect Gameserver.out and R3;
  signalroute R1 from env to Monitor with Newgame;
  signalroute R2 from env to Game with Probe,Result,Endgame;
  signalroute R3 from Game to env with Gameid,Win,Lose,Score;
  signalroute R4 from Game to Monitor with Gameover;
  process Monitor(1,1) referenced;
  process Game(0,) referenced;
endblock Game;
```

The specification given above starts with the keyword *block* and ends with *endblock*. Inside the body of the definition of the block *Game*, we start with the declaration of the internal signal *Gameover* by declaring its name, followed by the list of its parameters enclosed in parenthesis. The signal *Gameover* has a single parameter, the identification of a process (*Pid*) that is sending this signal.

Further on, we connect the channel *Gameserver.in* with the signaling paths R1 and R2. We also connect the channel *Gameserver.out* with the signaling path R3. We proceed with the declarations of signaling paths (keyword *signalroute*). Each declaration indicates the signaling path name, its direction (by using the keywords *from* and *to*), the names of the processes it connects (note that *env* is the special process which represents the environment), and a list of signals it carries (by using the keyword *with*). For example, the first signal path declaration shown in SDL-PR above declares the signaling path R1, which carries the signal *Newgame* from the process *env* (environment) to the process *Monitor*.

We end the definition of the functional block *Game* by declaring the processes it contains. A process in general is declared by the keyword *process*. A process declaration indicates the name of the process followed by the initial and maximal number of process instances that can appear in the system. The maximal number of process instances is an optional parameter, i.e., it can be omitted.

The process *Monitor* is declared as *Monitor(1,1)*, which means that the block *Game* should initially create one instance of this process and, at the same time, it is also the maximal number of *Monitor* instances that can be created in this block. Alternately, the process *Game* is declared as *Game(0,)*, which means that initially there are no *Game* instances, but also that the maximal number of Game instances is not limited, i.e., in theory it is allowed to create an infinite number of process *Game* instances inside the functional block *Game*. Of course, in reality this number is always limited to the available hardware resources.

In this particular example, we have declared two processes, *Monitor* and *Game*, that operate inside the functional block *Game*. The process *Monitor*

handles the interaction with a player. It is a mediator between the player and the process *Game*, which is essentially a model of the win–lose game. Due to the fact that the process *Monitor* is trivial and actually insignificant for this example, we will define only the process *Game* on the next hierarchically lower level of abstraction. On this level of detail, the process *Game* is modeled as a finite state machine (automata instance).

As already mentioned, the creators of SDL-GR (the graphical form of SDL) have extended the basic set of traditional flowchart symbols with a set of graphical symbols specialized for modeling finite state machines. The complete set of graphical symbols available for describing processes in SDL-GR is shown in Figure 3.36.

The meaning of the individual graphical symbols shown in Figure 3.36 is as follows:

- *State*: Specifies a stable state in which a process is blocked while waiting for one of the recognizable signals (referred to as *input*).
- *Input*: Specifies the reception of a given input signal (i.e., the occurrence of a certain event).

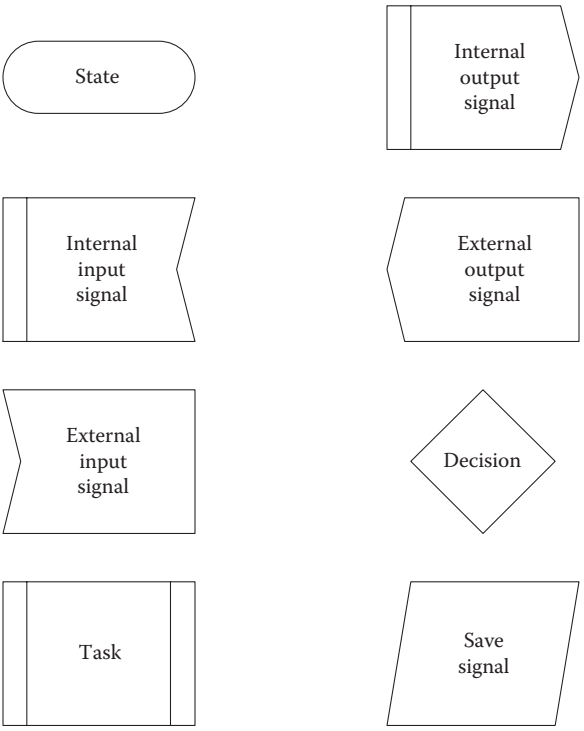


FIGURE 3.36
Set of graphical symbols available in SDL-GR.

- *Output*: Specifies the transmission of a given output signal (normally the output signal generated by a certain process represents an input signal for a process that receives it).
- *Decision*: Specifies an operation that checks if a given condition is true or false and, based on the outcome, selects one of the two possible paths in the current state transition.
- *Task*: Specifies an action in the course of current state transition that is neither *decision* nor *output.*
- *Save signal*: Specifies that recognition (processing) of a given signal should be postponed until it reaches a state where it is recognizable This symbol is used in specifications of signaling systems (e.g., SS7). It is seldom used in other applications, such as call processing.

The specification of a process in SDL-GR is generally made as a combination of the instances of the graphical symbols shown and explained above. An example of this type of specification is shown in Figure 3.37. It specifies and describes the process *Game*, the core of the win–lose game.

The evolution of the process starts from an unnamed state in the upper right corner of the graphical presentation (Figure 3.37). Starting from this state, the process unconditionally transits to its next stable state *even*. During this transition, the process *Game* sends the signal *Gameid* to the player.

While the process *Game* is in its stable state *even*, it awaits one of two possible events: the reception of the signal *Probe* or the expiration of the timer labeled *none*. If the timer expires, the process *Game* receives the corresponding signal *none*, and this causes the process to evolve into the next stable state *odd*. If the process receives the signal *Probe*, it sends the signal *Lose* to the player and updates the player's score, which is stored in the variable *count*, by adding one negative point. The process does not change its stable state, i.e., it remains in its current state (which is denoted with the character "–"), and that is the state *even*.

In its stable state *odd*, the process *Game* recognizes two same possible events, the reception of the signal *Probe* or the expiration of the timer labeled *none*. Actually, the timer *none* determines the time interval the process will spend in either the *even* or *odd* state before switching to the other one. Hence, if the timer *none* expires, the process evolves into the stable state *even*. Alternatively, if the process receives the signal *Probe*, it sends the signal *Win* to the player and updates the player's score (value of the variable *count*) by adding one positive point. The process remains in its current state (i.e., the state *odd*).

The upper left corner of the graphical representation of the process *Game* (Figure 3.37) shows one important example of simplifying SDL-GR diagrams. Because the reception of the input signals *Result* and *Endgame* is possible in both *even* and *odd* states, a straightforward solution would be to mechanically add these inputs and their processing to both states. The result would be a diagram that is much more complex and harder to understand

and follow. A more elegant solution is to draw the description of the processing of the inputs *Result* and *Endgame* in both states as a separate drawing in the diagram, as shown in Figure 3.37.

Generally, it is always useful to try to find identical processing of input signals (state transitions) that repeat in a number of stable states and to

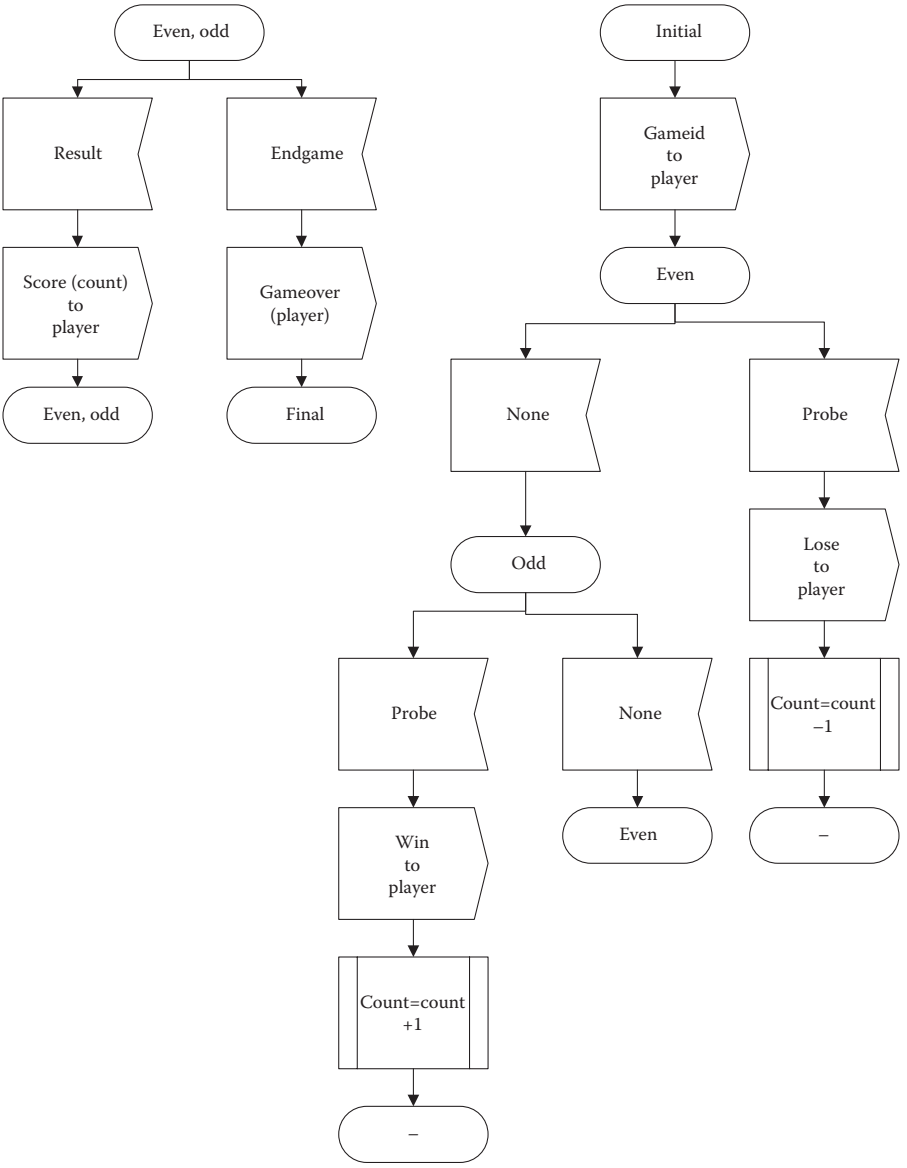


FIGURE 3.37
Process *Game* specification in SDL-GR.

simplify the specification by drawing these parts separately in the diagram. This type of a model reduction is really easy. We just draw an oval state symbol and write a list of the states (the list comprises the state names separated by commas) that share the common inputs inside the state symbol. Then we can copy and paste common state transitions. At the end, we can just remove the redundant state transitions. Of course, in the simple diagrams such as in the example at hand, we can see this in advance and draw accordingly, as we did for the processing of the inputs *Result* and *Endgame* in the states *even* and *odd*.

If the process *Game* receives the signal *Result*, which comes from the environment, i.e., from the player, the process sends the signal *Score(count)* to the environment (actually to the player) and it remains in its current state (*even* or *odd*). Alternately, if the process *Game* receives the signal *Endgame*, it sends the signal *Gameover* to the process *Monitor* and the game ends, i.e., the functional block deletes the process *Game*.

The specification of the process *Game* in SDL-PR (SDL program form) is as follows:

```
process Game(0,); fpar player Pid;
 dcl count Integer := 0; /* the counter that contains the result */
 start;
  output Gameid to player;
    nextstate even;
 state even;
  input none;
    nextstate odd;
  input Probe;
    output Lose to player;
    task count:=count-1;
    nextstate -;
 state odd;
  input Probe;
    output Win to player;
    task count:=count+1;
    nextstate -;
  input none;
    nextstate even;
 state even,odd;
  input Result;
    output Score(count) to player;
    nextstate -;
  input Endgame;
    output Gameover(player);
    stop;
endprocess Game;
```

The definition of the process starts with the keyword *process* and ends with the keyword *endprocess*. As already mentioned, initially no instances of the process *Game* are used, and the maximal number of its instances is unlimited. The process declaration is followed by the construct *fpar player Pid*, which defines the formal process parameter *player* that is assigned the value *Pid*. At the beginning of the game, the run-time environment creates an instance of the process, and assigns a unique *Pid* number to it.

Next, we declare the integer variable *count* (using the keyword *Integer*), which contains the current total value of points that the player has scored so far. After the label *start*, we define a series of statements that are executed by the process at its startup. In this example, the process *Game* at its startup sends the signal *Gameid* to the player and enters its initial stable state *even* (next state of the process is defined by the keyword *nextstate*).

For each stable state (keyword *state*) of the process, we define all the recognizable input signals (using the keyword *input*) and on the next level of indentation, we define the corresponding state transition as a series of statements that ends with the *nextstate* statement. For example, the recognizable input signals in the stable state *even* are the signal *none*, which relates to the expiration of the corresponding timer, and the signal *Probe*, generated by the player's stroke of the pushbutton. In the case the timer *none* expires, the process evolves to its next stable state *odd*. Alternatively, if the process receives the signal *Probe*, it sends the signal *Lose* to the player (using the keyword *output*), performs the task of decrementing the score by 1 (using the keyword *task*), and remains in its current state (the statement *nextstate -;*).

The stable state *odd* is defined in a similar manner. The input signals recognized by the process in its stable state *odd* are the signal *Probe* and the expiration of the timer *none*. If the process receives the signal *Probe*, it sends the signal *Win* to the player, increments the score by 1, and remains in its current stable state as *odd*. Alternatively, if the timer *none* expires, the process evolves into its stable state *even*. Finally, we define the state transitions initiated by the reception of the input signals *Result* and *Endgame* in either the state *even* or *odd*.

Understanding the principals of SDL-PR helps in more easily understanding the communications protocol software implementation in the state-of-the-art, higher-level programming languages such as C/C++ or Java. Although SDL-PR can resemble a pseudolanguage when compared to these programming languages, in reality it is a specialized language of higher level abstraction and it is feasible to construct a compiler for it. However, the study of the compilers is out of the scope of this book. The primary goal of this book in this respect is to provide an insight into the manual coding of SDL graphical diagrams in some of the abovementioned programming languages (C/C++ or Java).

The example under study can help in this respect. Obviously, two levels of nesting are included in it. The first level of nesting corresponds to the current stable state, in which the process is blocked while waiting for the next input signal, i.e., *start*, *even*, and *odd*. The second level of nesting corresponds to the type of input signal, i.e., *Probe*, *none*, *Result*, or *Endgame*.

The simplest method to implement this selection construction with two levels of nesting in the C/C++ or Java programming language is to use nested *switch-case* statements. The first *switch-case* statement is used to locate the current state. Then in each *case* clause of the first *switch-case* statement, another *switch-case* statement is used to locate the state transition statements

that correspond to the given input signal. This type of protocol implementation will be covered in detail in the next chapter.

3.7.1 Telephone Call Processing Example

The second example of the system specification made in SDL-GR is the specification of the telephone call processing system. The description of this system is given in the separate ITU-T recommendation Q.71. The Q.71 compliant program system consists of six mutually interconnected functional entities (referred to as functional blocks), namely FE1, FE2, FE3, FE4, FE5, and FE6 (Figure 3.38). The aim of this example is just to illustrate SDL-GR applicability, and the details of the recommendation Q.71 (such as the concrete names of the entities, their types and links, i.e., relations) are not really significant for the comprehension of the usage of SDL-GR. The reader that is more interested in Q.71 details can refer to the corresponding ITU-T recommendation. We use the hypothetical telephone call processing system *CallProcessor* to make further illustrations more concrete, without diving into the bulk of details of Q.71 recommendation. Comparing it to the real Q.71-compliant system, the *CallProcessor* is a very simplified academic example that consists of a single functional block, namely *TelephoneLine* (Figure 3.39). This functional block is linked with the environment by one input channel, named *input*, and one output channel, named *output*. So far, this example is very similar to the previous example *Daemongame*, which also comprises the single functional block *Game* that is interconnected with the environment with one input and one output channel.

The functional block *TelephoneLine* is shown in Figure 3.40. This simple functional block consists of the single process *FE1*. Two lists of signals are

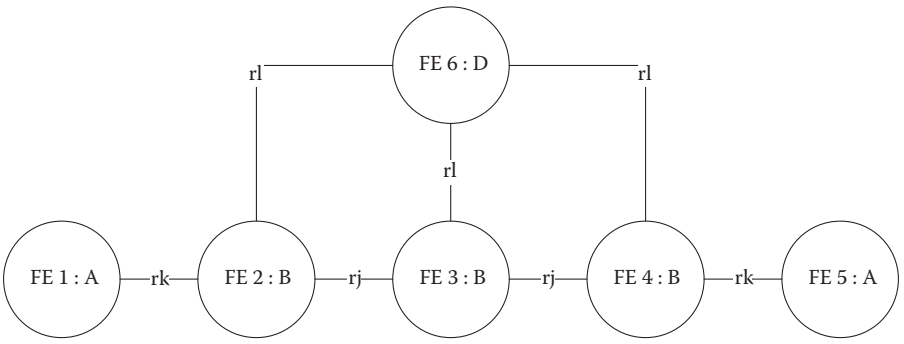


FIGURE 3.38
Functional model of the telephone call processing system.

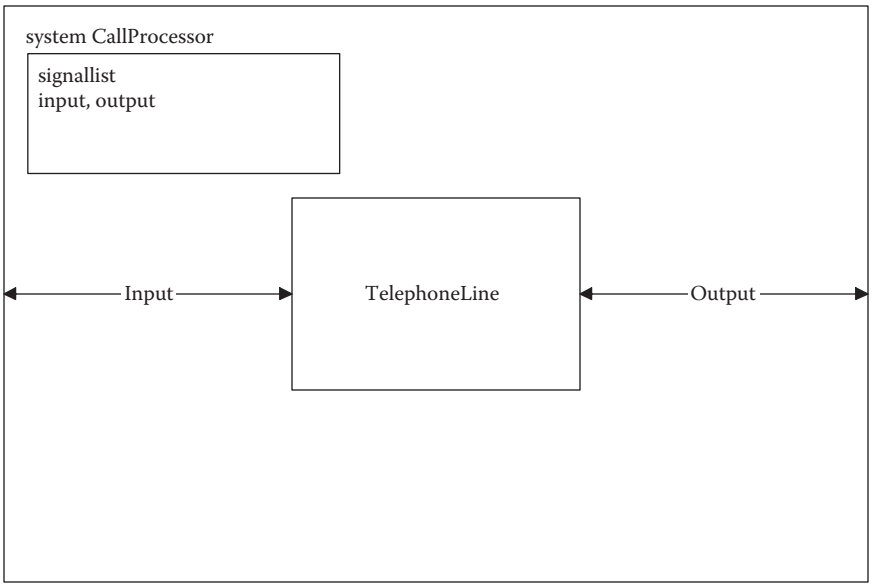


FIGURE 3.39
Hypothetical system *CallProcessor*.

declared (using the keyword *signallist*) in the upper left corner of Figure 3.39, namely, *input* and *output*. The process *FE1* is connected both to the telephone user (shown by the arrows placed at the right of *FE1*) and to the telephone exchange (indicated by the arrows placed at the bottom of *FE1*). It can receive one of the three possible input signals (*hookOff*, *dialDigit*, and *hookOn*) from the telephone user's side. Alternately, it can send the output signal *initiateOutgoingCall* to the telephone exchange or it can receive the input signal *asnwerReceived* from the exchange.

The process *FE1* is specified in the graphical form of SDL, SDL-GR, in Figure 3.41. This process resides in the telephone exchange and it communicates with the human that uses the telephone to establish a call, talk to the called party, and release the call at the end of the conversation. In reality, such a process must handle many scenarios, e.g., the user picks up the receiver but does not dial the number, or stops after dialing an insufficient number of digits.

The process specified in Figure 3.41 is rather simplified but it still captures the most significant part of the telephone line functionality on the calling party side. The telephone line in this context is a processor that hosts *FE1*, together with the interfacing hardware that connects it to both the calling party user's telephone and switching unit of the telephone exchange. For brevity, we refer to the former simply as a user and to the latter as a telephone exchange, or just an exchange.

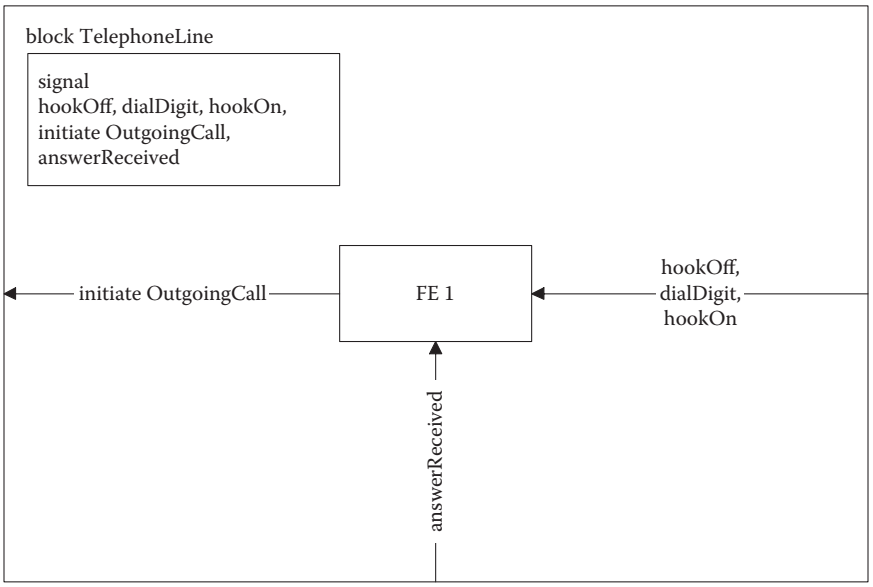


FIGURE 3.40
Structure of the functional block *TelephoneLine*.

The process *FE1* has four stable states, namely, *IDLE*, *WAIT_DIGIT*, *WAIT_ANSWER*, and *CONVERSATION*. The evolution of the process starts from the state *IDLE*. The single recognizable input signal in this state is the signal *hookOff*. If the process *FE1* receives the signal *hookOff*, it performs the task *prepareForDialing* and moves to its next stable state *WAIT_DIGIT*. While performing the task *prepareForDialing*, the process connects the free-to-dial tone to the calling party user. This tone serves as the indication to the user that they can start dialing the number of another user to which they wish to talk.

Two recognizable input signals are used in the stable state *WAIT_DIGIT*, i.e., the process can either receive the input signal *hookOn* or the input signal *dialDigit*. In this simplified example, we assume that the telephone number of the called party consists of a single digit. However, in real ISDN telephone networks, a so-called *enblock* dialing mode exists in which the ISDN terminal sends the complete telephone number to the telephone exchange in a single *SETUP* message. Therefore, this simplified example is not so far from reality. If the process *FE1* receives the input signal *hookOn*, it evolves into its initial state *IDLE*. If it receives the input signal *dialDigit*, it sends the output signal *initiateOutgoingCall* to the telephone exchange and it moves to the stable state *WAIT_ANSWER*.

In the stable state *WAIT_ANSWER*, two events are again possible—the reception of the input signal *hookOn* or the reception of the input signal *answerReceived*. In the former case, the process goes back to its initial state *IDLE*,

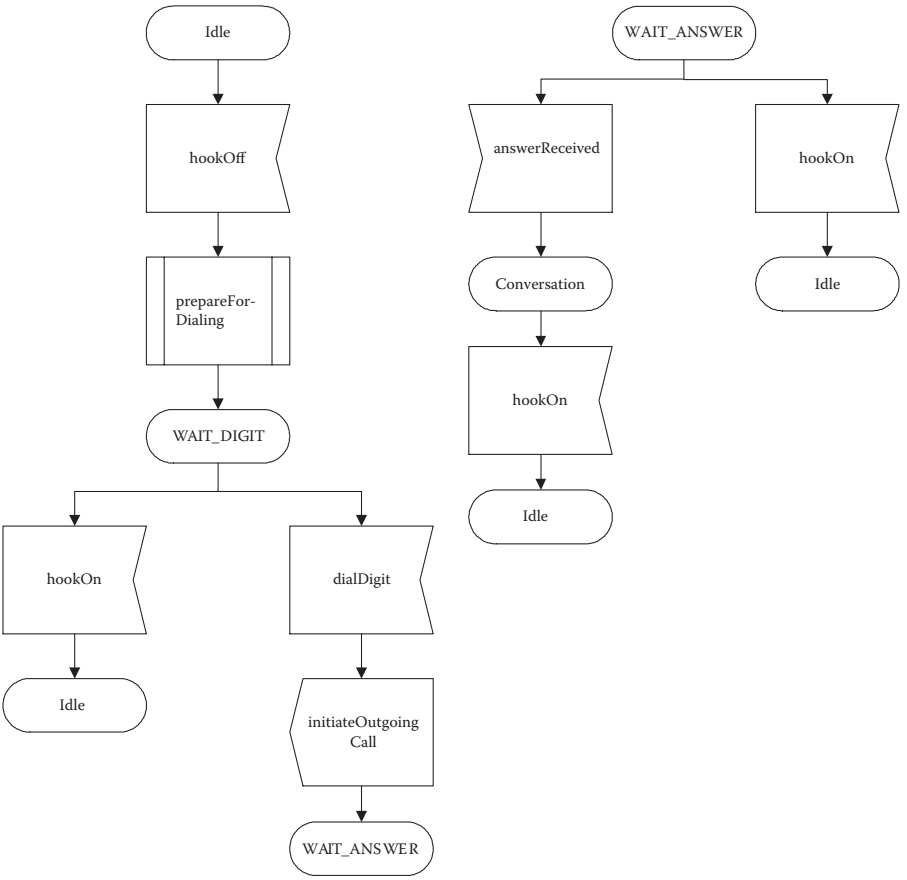


FIGURE 3.41
Simplified model of the Q.71 FE1 in SDL-GR.

whereas in the latter it evolves into its next stable state *CONVERSATION*. The input signal *asnwerReceived* is actually the result of the series of events that start with the input signal *hookOff* at the called party side. The telephone line entity at the called party translates it to the signal *answerIncomingCall* and sends it to the exchange at the called party side, which in turn sends it to the exchange at the calling party side. Finally, the exchange at the calling party side translates it to *answerReceived* and sends it to *FE1*.

In the final stable state *CONVERSATION*, only a single event is possible. The process *FE1* can receive the input signal *hookOff*, and if it does, that is the end of the conversation phase of the call and the process will return to its initial stable state *IDLE*. This closes the circle and the process is ready to process a new call originating from the same telephone line. Clearly, an instance of the process *FE1* is assigned to each telephone line in the telephone exchange.

In this example, we described the process *FE1* that is assigned to the calling party telephone line without going into a detailed specification of the operations performed by the telephone exchanges and the called party telephone line involved in the call. Obvious from this example should be that SDL diagrams are self-documented formal specifications and that no need really exists for any additional textual descriptions.

The SDL diagram shows the possible evolution paths of a process (a call processing in the example above). It defines unambiguously all telephone stable states, as well as all possible input signals for each state. The functional specification is based on the logical advance of a call, expressed in terms of telephony events. This makes it completely independent of both the hardware structure of the hosting system and the selected programming language and framework.

The SDL diagram is drawn based on the observations of a single telephone call without thinking about other calls, which are processed simultaneously (quasi-parallel by a single CPU or genuinely parallel by a multi-CPU system). This approach greatly simplifies software design. Finally, the existing SDL diagram can be easily extended by adding new states and input signals without the need to start drawing a new diagram from the very beginning. This possibility also enables the easy removal of revealed design errors.

3.8 Message Sequence Charts

An alternative method of specifying communication systems is by drawing message sequence charts that show the sequences of messages (signals) exchanged by the communicating entities. The ITU-T has developed a special language for this purpose, briefly referred to as MSC (Message Sequence Charts), and has standardized it in the Z.120 series of ITU-T recommendations.

MSC is based on the idea of following a single evolution path of a process. We start from a certain, most frequently initial, state of the process (e.g., the state *IDLE* in the previous example). After that, we select one of the possible input signals and follow the evolution path to which it points. In the previous example, a single input signal can be received in the state *IDLE*: signal *hookOff*, which causes the transition to the state *WAIT_DIGIT*.

In the newly reached stable state, we select again one of the recognizable events (the input signals that may be received in the stable state *WAIT_DIGIT* are *hookOff* or *dialDigit*; let us assume that we have selected *dialDigit*) and we follow the process evolution along the corresponding path (in the case of the input signal *dialDigit*, the process moves to the state *WAIT_ANSWER*). At the same time, as we mentally follow the evolution path of the process, we draw the messages that are exchanged between the process and its environment on the paper or, even better, the corresponding graphical editor.

The messages are represented by the graphical arrow symbols that are labeled by the message names. This is how we get the MSC charts.

Clearly, an MSC chart represents a single trace over the corresponding path, through the SDL diagram, or some other form of specifying finite state machines. We can see intuitively that for the real automata that we come across in practical applications, a finite number of paths exist that cover the SDL diagram. The set of the MSC charts that are obtained by visiting these paths represents the specification that is in a logical sense equivalent to the SDL diagram.

However, an obvious disadvantage of this type of a specification, in a form of a set of the MSC charts, is that it is much less evident than the SDL diagram. Therefore, when communication protocol designers refer to the formal specification, they really assume the SDL diagram. This disadvantage becomes obvious if, instead of dealing with a single automaton, we try to follow the evolution of a group of automata, which communicate between themselves, as well as with the environment, e.g., the group of automata defined in the abovementioned recommendation Q.71. The number of evolution traces of such systems can be extraordinarily large.

Not only must we select the initial state of a single automata, we must do it for all the automata from the group we want to analyze. Furthermore, in the case of simple and loosely coupled automata, an increase in the number of possible path combinations is not so high, but in the case of complex or tightly coupled automata, it is clear that the number of evolutions of the system can be huge.

The discussion above naturally raises the following questions: For what purpose are MSC charts useful? Do we need them at all? Practical experience shows that making the MSC charts can be useful at the beginning of the design process, when the designers talk rather freely about possible communication scenarios. These scenarios of message exchange most frequently represent the so-called main branches, i.e., main paths, through the protocol. Typically, they go from the beginning (the initial state) to the end (logically, the last state in the chain of states), e.g., from the state *IDLE* to the state *CONVERSATION*, such as in the previous example, without any errors or other exceptional events. Later, after finishing the analysis of the main paths, the paths of minor importance are analyzed. These are related to various less frequent cases, such as handling timer expirations, error recovery procedures, and so on.

All these scenarios, in the form of MSC charts, would be very useful in the later stages. Actually, these charts will be used as individual test cases during the implementation phase to partially check the functionality of the individual software modules (this is the so-called unit testing). They are also used during the final phase of the software verification as test cases for the compliance testing. The goal of **compliance testing** is to check if the software is compliant with the specification.

In most cases, the number of manually written MSC charts is finite and not too large (on the order of a few hundred at most). Later, during the testing

and verification phase, automatically generating a much larger number of test cases would be an ideal way (logically equivalent to MSC charts) to check the system much more thoroughly. This testing most frequently takes the form of statistical usage testing, which enables quality engineers to estimate the software reliability without any previous knowledge about the system under examination.

As already mentioned, the MSC language—similar to the SDL language—has both the graphical and program form. The graphical form of the MSC language is more interesting than the program form for developing communications software. The next example illustrates the message exchange among the functional entities *FE1*, *FE2*, *FE3*, *FE4*, and *FE5*, in the case of the successful establishment and successful release of the ISDN connection between two subscribers. From this example, MSC is obviously useful for tracing the message exchange between more processes, which is not so easy and clear by looking at the set of corresponding SDL diagrams.

We start drawing the MSC chart by placing the rectangle graphical symbols that represent the communicating entities (i.e., processes) at the top of the chart sheet. The names of the entities are used to label these rectangular symbols. Next, we draw a vertical line from each rectangular symbol to the bottom of the sheet. After that, we enter a series of messages exchanged by the processes shown on the top of the chart. Each message (i.e., signal) is represented by the arrow symbol labeled with the message name. The arrow starts from the vertical line that represents the process sending the message and ends at the vertical line that represents the process receiving the message. The time advances in the direction from top to bottom of the sheet, i.e., the messages that appear on the top of the chart are exchanged before the messages that appear at the bottom of the chart.

An example of the MSC chart is shown in Figure 3.42. This example illustrates the scenario of successful establishment and release of the ISDN connection. The functional entities *FE1* and *FE5* are assigned to the calling and called party user, respectively. Initially, the functional entity *FE1* receives the signal *SETUP_req* from the environment (in reality, this signal is generated by the signaling system DSS1). After receiving the signal *SETUP_req*, *FE1* translates it to the signal *SETUP_req_ind* and sends this new signal to *FE2*. *FE2* forwards this signal to *FE3*, *FE3* forwards it to *FE4*, and finally *FE4* forwards it to *FE5*.

After receiving the signal *SETUP_req_ind*, the functional entity *FE5* immediately sends two signals, the signal *SETUP_ind* to its environment and the signal *REPORT_req_ind* back to *FE4*. The latter signal is forwarded from *FE4* to *FE3*, then from *FE3* to *FE2*, and finally from *FE2* to *FE1*. *FE1* translates this signal to *REPORT_ind* and sends the latter to its environment.

The acceptance of the call by the calling party is signaled to *FE5* by the signal *SETUP_resp*. *FE5* translates this signal to the signal *SETUP_resp_conf* and sends the latter over the chain of FEs back to *FE1*. *FE1* in its turn translates it to *SETUP_conf* and sends the latter to its environment. This is the final step

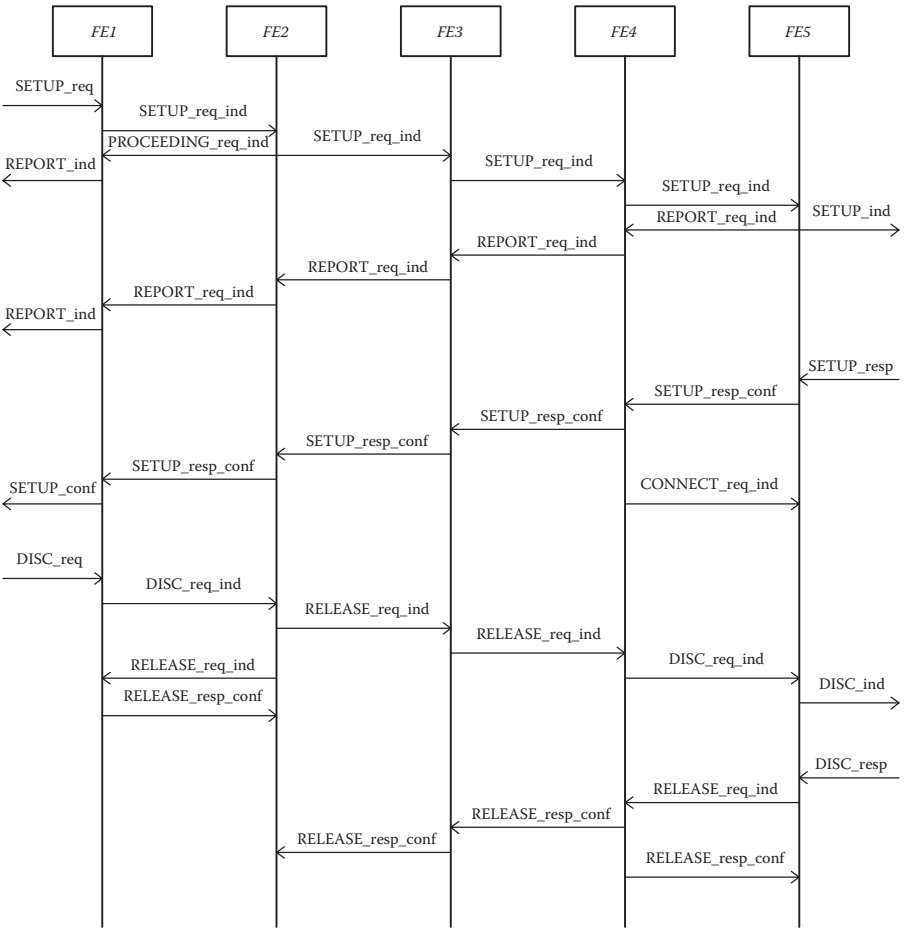


FIGURE 3.42
Example of the MSC chart: Successful ISDN call establishment and release.

of the connection establishment procedure. The next communication phase is a conversation.

At the end of the conversation, the calling party user initiates the call release procedure by sending the signal *DISC_req* to the functional entity *FE1*, which in turn translates it to *DISC_req_ind* and sends the latter to *FE2*. The functional entity *FE2* translates this signal to the signal *RELEASE_req_ind* and sends the latter to both *FE1* and *FE3*. From there, we have two parallel flows of messages. *FE1* replies to the signal *RELEASE_req_ind* by the signal *RELEASE_req_conf*. Alternately, *FE3* forwards the signal *RELEASE_req_ind* to *FE4*, which translates it to *DISC_req_ind* and sends the latter to *FE5*. *FE5* indicates the reception of that signal by sending the signal *DISC_ind* to its environment.

The environment answers with the signal *DISC_resp*, which is then translated to *RELEASE_req_ind* and sent to *FE4*. The functional entity *FE4* translates that to the signal *RELEASE_resp_conf* and sends the latter to both *FE3* and *FE5*. Finally, *FE3* forwards that final signal to *FE2*. This is the final step of the call release procedure.

This real-world example shows the main advantage of using MSC charts—instead of speculatively analyzing the parallel work of five finite state machines (*FE1*, *FE2*, *FE3*, *FE4*, and *FE5*) by looking at their SDL diagrams, here on a single chart we see how the system evolves through the procedures of call establishment and release. At this level of abstraction, we are not interested in the individual work of the individual automata. We just follow the interaction based on the message exchange between the automata in a given group.

3.9 Tree and Tabular Combined Notation Version 3

In this section, we cover the Testing and Test Control Notation Version 3 (TTCN-3), a language that was originally standardized by the European Telecommunication Standardization Institute (ETSI) by extending the previous language Version 2 (TTCN-2). A group of designers may employ TTCN-3 to make a formal specification of test procedures that are used to check if the implementation behaves in conformance with the system's formal specification. The type of testing that is conducted in accordance with such test procedures is referred to as **conformance testing**. The object of the testing is typically called an **implementation under test** (IUT) or a **system under test** (SUT). Since an IUT might be a part of a SUT, in this section we use the term SUT as a more general term. The system that is used to test a SUT is called the **test system** (TS), see Figure 3.43.

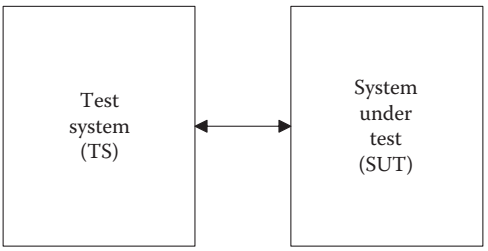


FIGURE 3.43
Standard test configuration.

Since TTCN-3 is a rather complex standard, here we cover the TTCN-3 basic features, which are sufficient for making simple test suites, and we leave the more advanced TTCN-3 features (such as multicomponent TTCN-3 and procedure-based communication) to the reader as an option for further study (see Willock, 2011). Therefore, this section is organized in the following subsections:

- TTCN-3 Language, Test Suite, and Test Systems
- Basic TTCN-3 Constructs and Statements
- Single Component TTCN-3 Test Suites

3.9.1 TTCN-3 Language, Test Suite, and Test Systems

TTCN-3 is an internationally standardized language specially designed for testing. Besides reusing many basic constructs and statements from conventional programming languages, TTCN-3 introduces testing-oriented extensions and more advanced concepts, including (1) the type system with native types for lists, test verdicts, and test components; (2) direct support for timers, message-based and procedure-based communication; and (3) built-in data matching, distributed test system architecture, and test components concurrent execution.

TTCN-3 standards provide clear and precise language definitions, so test cases written in TTCN-3 are unambiguous, and their execution on any TTCN-3 compliant testing system must have the same behavior. This independence from testing tool vendors enables easy test suite migration to other testing tools and their reuse. Although in this book, we primarily use TTCN-3 for conformance testing, actually TTCN-3 may be used across the whole product development cycle.

The TTCN-3 core notation is an intuitive textual format for defining test cases that is quite similar to conventional programming languages. Additionally, TTCN-3 supports specifying test cases using other presentation formats. These presentation formats may be converted into the core notation with the same semantics. Initially, two presentation formats have been standardized, namely the tabular presentation format and the graphical presentation format. The former was designed to further the support of the existing TTCN-2 tabular format, and enable migration of existing TTCN-2 legacy artifacts into the TTCN-3 tools. The latter uses an extended version of the MSC notation for specifying test cases. Since neither of these two presentation formats were accepted by the TTCN-3 community, they are not described further in this book.

When comparing TTCN-3 with TTCN-2, we need to consider the four major areas of improvement, namely the productivity, the expressiveness, the flexibility, and the extensibility. The core TTCN-3 notation has been developed as a textual language resembling conventional programming languages,

with the intention to enable productivity. TTCN-3's better expressiveness and flexibility is based on various language extensions, such as (1) support for the testing IP based systems and text based protocols like HTTP and SIP, and (2) support for testing systems based on remote procedure calls, like CORBA and web services. In order to support extensibility, TTCN-3 has explicit hooks and mechanisms that allow new language features and notations to be easily integrated.

Generally, there are two kinds of new features: the self-contained and the multifaceted features. Some examples of self-contained features are the integration of IDL and XML type definitions and the definition of a common set of documentation tags, whereas some examples of multifaceted features are behavior types, type parameterization, and test deployment support. The self-contained features have been defined by the new parts in the TTCN-3 standards, whereas the multifaceted features have been defined by separate extension packages, including the necessary modifications to the core language, the operational semantics, and the parts of the TTCN-3 standards.

Next, we introduce the TTCN-3 Test Suite. The TTCN-3 test suite is a collection of modules. A **module** comprises definitions of data types, values, and test cases, as well as a control part that specifies how different test cases are to be executed. A module may import necessary definitions from other modules, which is key for test suite modularization.

Obviously, a test case running on a test system must be able to communicate with a SUT in order to test it. Generally, TTCN-3 supports two types of communication among test cases and SUTs, namely message-based communication and procedure-based communication. Since message-based communication is still dominantly used, we focus on it in this book. Normally, we use the TTCN-3 type **record** to define needed message types. A record is an ordered structured type, which is a collection of basic type elements (such as **integer** and **charstring**) and other structured type elements that correspond to individual message fields. Many message types comprise a field that defines the kind of the message, and such a field is typically defined using the TTCN-3 type **enumerated**.

To define a message type, we normally first define the types of individual message fields, and then we define the message type itself. For example, let's define the message type *Msg*, which comprises the four fields, namely *ID*, *Kind*, *Question*, and *Answer*. Assume that the types of these fields are 16-bit integer, enumerated, charstring, and charstring, respectively. We would then use the following definitions to define the message type *Msg*:

type integer *ID* (0..65535);
type enumerated *Kind* (*e_Question*, *e_Answer*);
type charstring *Question*;
type charstring *Answer*;
type record *Msg* {

```
  ID id;
  Kind kind;
  Question question;
  Answer answer;
}
```

The communication messages are the actual instances of message types, and these instances are called **templates**. In TTCN-3, templates are used to send particular messages or to test whether received messages are in the set of expected messages. We may specify a set of expected messages using ranges, lists, and matching attributes (we will illustrate this later on). It is important to remember that a template for a type must specify a value or a matching expression for each field of this type. If the value of some field type is unknown, i.e., it may contain any value, we encode such a value with the character "?." Furthermore, if some field of a message type is optional, and if this field should not be the part of the template we are creating, then we have to assign the special value **omit** to this field.

The template definition resembles the definition of a function. We specify the type (**template**), the name, and the list of its formal parameters. Instantiating a template (i.e., creating the concrete message) resembles a function call—we specify the template name and the list of its real parameters, which are used to replace the formal parameters.

For example, using the message type *Msg*, we may define the parametrized template *t_request* as follows:

```
template Msg t_request( ID p_id, Question p_question) := {
  id := p_id,
  kind := e_Question,
  question := p_question,
  answer := omit
}
```

Similarly, we may define the parametrized template *t_response*:

```
template Msg t_response( ID p_id, Answer p_answer) := {
  id := p_id,
  kind := e_Answer,
  question := ?,
  answer := p_answer
}
```

Next, we introduce test **components** and communication **ports**. Each test case **runs on** the test component it has been assigned to. A test suite may use a single or multiple test components, which may communicate with each other, and/or with the SUT over communication ports. A port is theoretically

an infinite first-in-first-out (FIFO) queue oriented in the receive direction, which stores messages in message-based communication (or calls in procedure-based communication) until they are processed by the test component that is the owner of that port. Many simple test suites use a single test component to execute test cases, and a single communication port to communicate with the SUT. Here is an example definition of the test component named *ComponentS*, which uses the single port *pt* of the type *PortS*:

```
type component ComponentS {
  port PortS pt
}
```

Each port has a type, which defines the type of the communication (message-based or procedure-based) and the types of messages that may be communicated over that port. Within the definition of a port, each message type is given the attribute **in/out/inout** that determines whether a message of that type may be received (**in**), sent (**out**), or received and sent (**inout**) from the test component owning that port. For example, the port type *PortS* that may be used to both send and receive the message type *Msg* is defined as follows:

```
type port PortS {
  inout Msg
}
```

A test case may send and receive messages (i.e., templates) on a port using the method **send** and the method **receive**, respectively. For example, a test case sends the message *t_requestMsg* on the **out/inout** port *pt* using the following statement:

```
pt.send( t_requestMsg(12345, "SUT what is your name?") );
```

Similarly, a test case receives the message *t_responseMsg* on the **in/inout** port *pt* using the following statement:

```
pt.receive( t_responseMsg(12345, "My name is SUT XY.") );
```

Although the syntax of statements for sending and receiving messages is very similar (only the method name is different), there is a fundamental difference in their semantics, i.e., in the way they operate. The method **send** is a nonblocking method and it always successfully sends the outgoing message, whereas the method **receive** is a blocking method and returns when the specified incoming message appears at the head of the input FIFO queue. More precisely, if the input FIFO queue is empty, the method **receive** blocks until the specified message arrives into the input FIFO queue. If some other message is at the head of the input FIFO queue, the method **receive** remains

blocked forever (we normally use timers to recover from such situations, as will be shown later).

A typical test case resembles a single- or multiphase interview. In each phase, the test case asks a question by sending a request message to the SUT, and then it checks the SUT's answer by matching the SUT's response message with the expected message (i.e., particular template). If the response matches the expected message, SUT passed that phase, and the test case proceeds to the next phase. If SUT passes all the phases, the test case sets the final verdict by using the keyword **setverdict** to the value **pass**. An example of the body of a simple single-phase test case is the following:

```
pt.send( t_requestMsg(12345, "SUT what is your name?") );
pt.receive( t_responseMsg(12345, "My name is SUT XY.") );
setverdict(pass);
stop;
```

In this simple example above, the test case sends the messages *t_requestMsg* over the port *pt*, matches the SUT's response form the same port *pt* to the message *t_replyMsg*, and if the test case receives that message, it sets the verdict to **pass**, and stops its execution using the keyword **stop**.

Besides the verdict **pass**, the test verdict may be **none**, **inconc** (i.e., inconclusive), **fail**, or **error**. The meaning of these verdicts are as follows (we will return to more detailed technical treatment of test verdicts later in the text that follows):

- The verdict **none** is the default verdict and it is implicitly set by the test system before the test case starts executing.
- The verdict **pass** means that the test case has been completed successfully.
- The verdict **inconc** means that there is not enough evidence to proclaim that the SUT is conformant to the specification.
- The verdict **fail** indicates that the SUT is not compliant with the specification.
- The verdict **error** indicates that the test case terminated with a runtime error, e.g., divide by zero.

In order to complete the simple test case above, we have to give it a name and define the test component that will execute it. If we give it the name *tc_simple1* and if we assume that it will run on the component *ComponentS*, the complete test case would be the following:

```
testcase tc_simple1() runs on ComponentS {
  pt.send( t_requestMsg(12345, "SUT what is your name?") );
  pt.receive( t_responseMsg(12345, "My name is SUT XY.") );
```

```
  setverdict(pass);
  stop;
}
```

If we want to activate this test case, we have to declare that it should execute, by using the keyword **execute**, within the control part of the test module, which is declared by the keyword **control**, as follows:

```
control {
 execute( tc_simple1() )
}
```

Although our test case above looks simple and elegant, it will be blocked forever in two cases. The first case is when the SUT sends the unexpected response message, i.e., when it sends any other message not equal to *t_responseMsg*(12345, "My name is SUT XY."). The second case is when the SUT, for some reason, does not send any response message at all. So, besides the successful case when the SUT returns the expected response message within some reasonable amount of time, we have two failure cases.

Generally, in TTCN-3 we use the statement **alt** to specify alternative SUT behaviors at a given point of a test case. The statement **alt** blocks until any of its alternatives matches. The alternatives are checked, starting from the first and towards the last, until the first matching alternative is found. The way to receive any message is to use the method **receive** without parameters, which will match with any message at the head of the input FIFO queue.

We may fix the initial test case above by introducing the statement **alt** with three alternatives, which correspond to three possible SUT behaviors (i.e., the one successful and the two failure cases). Additionally, we must introduce a timer that will limit the time interval for awaiting a response message from SUT. Let's give this timer the name *responseTimer*. This timer should be started before the statement **alt**, and it should be stopped when any response message from SUT is received. Of course, there is no need to stop the expired timer.

The fixed test case operates as follows. It sends the request message to the SUT, starts the timer *responseTimer*, and checks the alternatives. If the response message is the expected one, it sets the verdict to **pass**. If the response message is any other (unexpected) message, it stops the time *responseTimer*, and sets the verdict to **fail**. If the timer *responseTimer* expires, it just sets the verdict to **fail**. The complete code of the fixed test case is as follows:

```
testcase tc_simple1() runs on ComponentS {
  timer responseTimer;
  pt.send(t_requestMsg(12345, "SUT what is your name?"));
  responseTimer.start(5.0);
  alt {
    []pt.receive(t_responseMsg(12345, "My name is SUT XY.")){
```

```
    responseTimer.stop;
    setverdict(pass);
   }
 []pt.receive {
   responseTimer.stop;
   setverdict(fail);
  }
 []responseTimer.timeout {
   setverdict(fail);
  }
 }
 stop;
}
```

Obviously, dealing with unexpected or untimely SUT behavior may lead to considerable code duplication. If we needed to add two additional cases for every receive statement in order to catch incorrect or missing responses, then our test cases would become very long and verbose, thus hard to comprehend and maintain. Because of this, TTCN-3 offers a so-called **default behavior** construct, which allows us to handle unexpected situations implicitly. Instead of writing code to handle unexpected situations explicitly, we write the default behavior handler in a single place and define that this handler should be used implicitly when none of the explicitly available alternatives match (we will come to this later in this section).

Sometimes, test cases that require access to more than one interface can be better structured by having one dedicated test component per interface. These interfaces need to be described within the so-called **test system interface** (TSI), which defines the common interface towards the SUT that different test components share in order to test the SUT. One of these components is called the **Main Test Component** (MTC), which is typically responsible for creating other test components, collecting their individual verdicts, and calculating the final verdict for the whole test case.

Next, we introduce TTCN-3 Test Systems. So far, we learned the TTCN-3 language elements for writing the so-called **abstract test suites**, which do not provide any system specific information, such as message encoding or practical communication setup. In the abstract test cases shown in this section, we send and receive messages without being concerned with the details how these messages are sent in the physical world. However, in order to create real test suite, we have to commute from an abstract to the real world. An abstract test suite is not directly executable, so we have to provide a TTCN-3 compiler or interpreter for it. Additionally, outside of an abstract test suite, we have to provide the following parts:

- **Message Codecs**, which are able to encode messages that are sent to the SUT and decode messages that are received from the SUT.

- An **SUT Adapter** that maps the TTCN-3 port to the real port used by the SUT, and the TTCN-3 communication mechanism to the real communication mechanism used by the SUT.
- A **Platform Adapter**, which typically provides the real implementation of timers and the mechanism for calling external platform specific functions.
- **Test Management** provides support for creating test campaigns, or for customizing log formats and handling log records. It is especially important for a dynamic test environment, where test cases and/or order of their execution are frequently changed. In this case, we need advanced test management support in order to avoid unnecessary, time-consuming test suite recompilations.

The additional parts, mentioned above, communicate with the abstract test suite using the two standard interfaces, namely the **TTCN-3 Runtime Interface** (TRI) and the **TTCN-3 Control Interface** (TCI). The TRI specifies operations for the SUT adapter and the platform adapter, whereas the TCI specifies operations for the test management, the component handling, the logging, and the encoders and the decoders (i.e., codecs). Figure 3.44 shows the block diagram of the complete TTCN-3 Test System architecture.

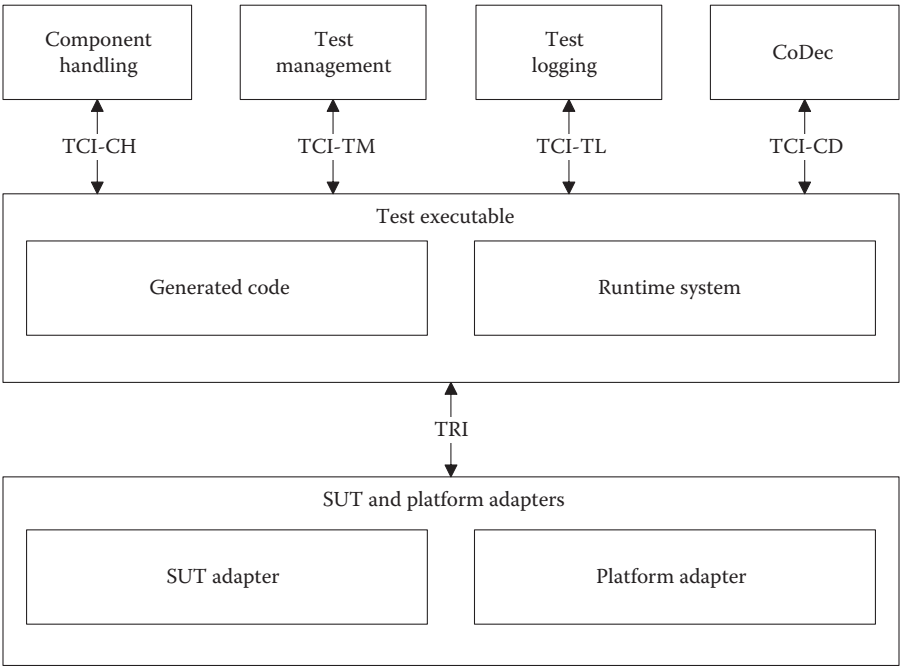


FIGURE 3.44
Architecture of the TTCN-3 Test System.

The source abstract test suite is compiled into the module **Generated Code**, shown in the center of the Figure 3.44. The generated code is executed on the module **Runtime System**, which implements the TTCN-3 operational semantics. These two modules together are called the **TTCN-3 Executable** (TE). The test executable uses the interface TRI to call functions provided by the **SUT Adapter** and the **Platform Adapter**, shown in Figure 3.44. These adapters map common operational abstractions, like communication ports and timers, to real mechanisms available on particular test system platforms.

On the other hand, the interface TCI connects the test executable with the rest of the modules shown in Figure 3.44, namely the **component handling** (CH), the **test management** (TM), the **test logging** (TL), and the **codec** (CD). Since the interface TCI is rather complex, it has been partitioned into the four sub-interfaces: the interface TCI-CH, the interface TCI-TM, the interface TCI-TL, and the interface TCI-CD. The roles of these four modules (and their corresponding sub-interfaces) are the following: The module CH is used to specify how test components are created and implemented when the test system is actually deployed, the module TM is used to control test case creation and execution, the module TL used to create execution logs, and the module CD is used to specify external codecs.

3.9.2 Basic TTCN-3 Constructs and Statements

In this section we briefly introduce basic TTCN-3 constructs and statements. The TTCN-3 test suite consists of modules like programs in common programming languages. Each module may have a definition part and an optional control part. A control part is similar to the function **main** in programming languages. In this section, we focus primarily on the module definition part.

The basic TTCN-3 constructs are the following:

- Identifiers
- Modules
- Scopes
- Constants
- Variables
- Comments
- Basic data types
- Subtypes
- Functions
- Predefined functions
- Parameters with default values

Identifiers uniquely identify named entities in the TTCN-3 code in the same way that identifiers in programming languages do. They consist of alphanumeric characters and underscores, must start with a letter, and are

case sensitive. TTCN-3 has its own naming convention for identifiers, which is rather similar to naming conventions in programming languages. So, we skip its formal specification here, and instead use it consistently in the code snippets in this section, so readers will become familiar with it.

Modules are defined using the keyword **module** followed by the module name and the module body, which is enclosed in the curly brackets. The module body consists of a definition part and an optional control part. The control part is defined using the keyword **control** followed by the control part body that is enclosed in the curly brackets. The body of the control part defines how the defined test cases are to be executed. The syntax for defining modules is the following:

```
module module_name {
  // Here goes the definition part, which defines data types and constants

  control {
  // Here goes the control part that executes the test cases
  }
}
```

Scopes are defined by code blocks enclosed in the curly brackets. The code blocks may contain code statements and nested code blocks. The outermost scope is the current module. The purpose of TTCN-3 scopes is the same as in programming languages, and they follow the same rules. Definitions made in the current scope are only visible within that scope and in the nested scopes. In TTCN-3, it is not possible to reuse the identifiers that were introduced in the outer scopes. The following are the nine basic scope units:

- Module definitions part
- Control part of a module
- Component types
- Functions
- Altsteps
- Test cases
- Statement blocks
- Templates
- User-defined named types

All the identifiers must be declared before they are used, except the module identifiers, which may be declared and referred to in any order.

Constants are defined using the keyword **const** in any scope. A constant is assigned the value within its declaration, which has the following syntax:

```
const const_type const_name := const_value;
```

where *const_type* is the type of the constant, *const_name* is the identifier of the constant, and *const_value* is the value assigned to the constant. Generally, *const_value* may be an expression with constants, but references to other constants must be made without creating cycles. By using constants, we create test suites that are easier to understand and maintain.

Variables are declared using the keyword **var** in any scope, except at the top module level, because in TTCN-3 there are no global variables. Global variables are not allowed in TTCN-3 because of data races that would otherwise occur when distributed test components would try to update them. Like in programming languages, variables are used to save temporary values during program execution. A variable may be assigned the initial value within its declaration, or later in a separate assignment statement. However, using a variable before it is assigned a value results in a run-time error.

Besides simple variables, we can declare arrays the same way we do in other programming languages, after the array name we define its size enclosed by square brackets. Arrays are indexed starting from 0, and any attempt to access a value outside of the permitted range would lead to an error.

Comments in TTCN-3 are classified as block comments, line comments, or documentation comments. The block comment starts with characters "/*", may span several lines, and ends with characters "*/", whereas the line comment starts with characters "//" and extends to the end of the line. Documentation comments are defined in standard ETSI ES 201 873-10 (see Part 10: TTCN-3 Documentation Comment Specification). Like in some other programming languages, such as Java, an external documenting tool processes these documentation comments to automatically generate the up-to-date test suite user documentation.

Basic data types, also known as built-in data types, are a constitutive part of the TTCN-3 language. The TTCN-3 may be classified as a strongly typed language with a very rich type system. Here we will introduce only the most frequently used simple data types and the subtyping mechanisms for introducing user-defined types. The most frequently used basic data types are **integer**, **Boolean,** and **charstring**. Possible values of the type integer are positive and negative whole numbers, including zero, possible values of the type **Boolean** are **true** and **false**, whereas possible values of the type **charstring** are strings of ASCII characters that are enclosed by double quotes. However, unlike in other programming languages, nonprintable control characters, such as new line or tab, cannot be expressed using escape sequences.

Subtypes in TTCN-3 may be defined using two available subtyping mechanisms. The first subtyping mechanism restricts the set of possible values of a given ordered type to a particular range of values. For example, the type i**nteger** may be subtyped to a range of its values, by specifying a lower and an upper bound of that range. According to the first subtyping mechanism, a new subtype is defined by the type declaration of the following syntax:

type *parent_type new_type new_type_range*

Here, *parent_type* is the parent type, *new_type* is the name for the newly defined type, and *new_type_range* is the new subtype's restricted range of values. We already saw the following example of the first subtyping mechanism in the previous section:

type integer *ID* (0..65535);

A constant or a variable of the given subtype must obey the subtype restrictions. An assignment outside of the allowed range of values would cause a compile time or run time error. For example, assigning the value -1 to a variable of the type *ID* would cause such an error.

The second subtyping mechanism restricts the set of possible values of a given ordered type to a particular list of values. According to the second subtyping mechanism, a new subtype is defined by the type declaration of the following syntax:

type *parent_type new_type new_type_list*

where *parent_type* is the parent type, *new_type* is the name for the newly defined type, and *new_type_list* is the new subtype's restricted list of values. For example, we define the new type *SomeNumbers* by listing the list of its possible values 1, 3, 5, and 8:

type integer *SomeNumbers* (1, 3, 5, 8);

While introducing subtyping, we already touch upon compatibility restrictions. TTCN-3 enforces type compatibility of values in assignments, instantiations, expressions, and comparisons. We already mentioned that assigning the value to the given variable that is outside of its set of possible values causes a compile time or a run time error. In principle, a variable can be assigned a value of another type if they have the same root type and the value conforms to the associated subtype constraints of that variable.

Functions are defined in the module definitions part by the keyword **function**, followed by a function name, an optional parameter list, an optional return value, and the function body enclosed by curly brackets. A function body typically contains definitions of local constants and variables, and statements that define dynamic behavior. Functions may be called from the module control part, from test cases, or from other functions.

The function's return value may be a value like in common programming languages or a template. The return value is defined by the keyword **return** after the parameter list in the function header, followed by the return type. In this case, the function body must contain at least one return statement followed by a value or template, which must be compatible with the specified type in the function header.

Function parameters are declared with an optional passing mode, their type, and their name. There are the three parameter passing modes, namely the passing mode **in** (this is the default mode), the passing mode **out**, and the passing mode **inout**. In case of the passing mode **in**, function parameters are passed by value, i.e., the actual parameters are copied into the formal parameters before the function body is executed. In cases of the passing modes **out** and **inout**, function parameters are passed by reference. In particular, in the case of the passing mode **out**, the formal parameters are copied into actual parameters, whereas in the case of the passing mode **inout**, parameter passing is performed in both directions. Obviously, an actual parameter cannot be a constant if it is to be passed in the modes **out** and **inout**.

TTCN-3 introduces a term **instantiating** a function, which corresponds to a function call in other programming languages. We may instantiate a function by specifying the function name and its actual parameters. There are two possible ways to specify the actual parameters—with or without the parameter names. If actual parameters are specified without their names, they must be specified in the same order as the corresponding formal parameters that are specified in the function header. If the actual parameters are specified by referring to the names of formal parameters, they may be specified in any order.

As in other programming languages, functions may be defined externally, i.e., outside of the current module. We use the keyword **external** in front of the function prototype to declare such a function.

Predefined functions are functions prepared in advance that are already available for use, much like built-in (or basic) data types. These functions enable productive work—without them the user would need to write everything from scratch. The most important predefined functions are as follows:

- Value conversion functions, e.g., integer to a character
- String handling functions
- Length and size functions
- Presence checking functions
- Codec functions

Parameters with default values enable smooth evolution of test suite libraries by adding parameters without breaking previous releases. Once a formal parameter with a default value is defined in the list of formal parameters, it may be omitted in an actual parameter list. Obviously, an **in** parameter may have a default value, whereas **out** and **inout** parameters may not have a default value.

The parameter default value is used when no actual parameter is provided for a formal one. Typically, when the trailing formal parameters in a parameter list have default values, they can all be omitted in an actual parameter

list. In case of other parameters that follow a parameter with a default value, the parameter with the default value could be omitted in the actual parameter list by using the character dash '-' instead of a value.

The alternative convention of providing actual parameters is to assign actual parameters to the formal parameter names explicitly. This alternative convention may be used for all the parameter passing modes (**in**, **out**, and **inout**). However, it is not allowed to mix the conventional and the assignment conventions. Also, the assignment convention may not be used for the parameter with a default value.

Here we conclude our brief introduction to basic TTCN-3 constructs, and we switch to basic TTCN-3 statements. The basic TTCN-3 statements are

- Operators
- Expressions
- Assignments
- Conditional statements
- Loops
- Labels and goto statements
- Log statements
- Control part
- Preprocessing macros

Operators are classified into five categories: arithmetic operators (+, -, *, /, mod, rem), relational operators (==, <, >, != , >=, <=), logical operators (not and, or, xor), binary string operators (not4b, and4b, xor4b, or4b), and string operators (&, <<, >>, <@, @>). Operator precedence (i.e., operator priorities) is defined similar to other programming languages, e.g., / has higher priority than +, etc.

We construct **expressions** by applying operators to operands, which may be literals, constants, and variables. Expressions are evaluated according to operator priorities, or from left to right when operators have the same priority. If in doubt, we may group subexpressions by parentheses. Of course, operands of arithmetic, logical, and string concatenation operators must have the same root type. In TTCN-3, all variables must be initialized before the expression is evaluated (unlike common programming languages where this is not required).

Assignments are used to update variables. The expression on the right-hand side and the variable of the left-hand side must be of compatible types and the expression must evaluate to a value. If these conditions are met, the value of the expression is stored into the variable.

Conditional statements, like in other programming languages, are used to organize control flow within the dynamic parts of test suites. There are two kinds of conditional statements: the statement **if–else** and the statement

select–case–else. These statements may be nested and mutually nested. The syntax of the statement **if–else** is as follows:

```
if (condition_expression)
  statement_true
else
  statement_false
```

where *condition_expression* is a Boolean expression, *statement_true* is a statement that is executed if the expression evaluates to the value **true**, and *statement_false* is a statement that is executed otherwise. Most frequently, these statements are some block statements wherein some processing is performed:

```
if (condition_expression) {
  // Do something if the condition is true
}
else {
  // Do something else if the condition is not true
}
```

The syntax of the statement **select–case–else** is

```
select (control_variable) {
  case (values_1)
    statement_1
  ...
  case (values_n)
    statement_n
  ...
  else
     statement_else
}
```

where *control_variable* is the name of the control variable that governs the selection of possible cases, *values_1* to *values_n* are the specifications of possible values, *statement_1* to *statement_n* are the corresponding statements, and *statement_else* is the statement that is executed if none of the cases was selected. Here is a simple example:

```
integer v_int;
  // assume that a value has been assigned to v_int
  select (v_int) {
    case (0 .. 9) {
      log(v_int, " is a one digit positive integer");
    } case (10 .. 19) {
      log(v_int, " is a two digits positive integer");
```

```
 }else case{
   log( v_int, " is not a one digit or a two digits positive integer" );
 }
}
```

Loops are used to specify repetitive behavior. There are the three kinds of loops in TTCN-3: the statement **for**, the statement **do–while**, and the statement **while**. Within a loop, the statement **break** may be used to exit the loop, whereas the statement **continue** may be used to skip the current iteration. The syntax of the statement **for** is as follows:

```
for ( initial_stmt; condition_exp; next_stmt )
  body_statement
```

where *initial_stmt* is the initial statement typically used to declare a control variable and to assign it an initial value, *condition_exp* is the condition expression that is checked before the next loop iteration starts (if the expression is not **true**, the loop terminates), *next_stmt* is the statement that is executed after each iteration, and *body_statement* is the loop's body. The following simple example, with the typical control variable *i*, looks familiar:

```
for (integer i := 0; i < n; i := i + 1 )
  // Do something that depends on the value of i
```

The syntax of the statement **do–while** is as follows:

```
do
  body_statement
while ( condition_exp )
```

The syntax of the statement **while** is

```
while ( condition_exp )
  body_statement
```

Labels and goto statements provide a mechanism to jump from one part of a program to another. Although they provide compatibility with TTCN-2, their usage in TTCN-3 is strongly discouraged. The statement **label** defines a label within a logical block statement (e.g., function or control part), whereas the statement **goto** is a control flow statement that transfers control to the specified label within the same block statement. So, it is not possible to jump out of (or into) the functions, test cases, and the control part; it is not possible to jump into both the loop and conditional statements.

Log statements are used for writing relevant information on the test system's logging interface. The particular format of logged values depends

on the logging interface implementation. We may log the variables, arrays (whole arrays by specifying their name), constants, function parameters, function instances (that have the statement **return**), test component references, templates, timers, and related operations.

The **control part** of the module is the entry point for execution of a test suite, which is similar to the function **main** in other programing languages. The control part specifies the dynamic behavior of the test system. It may contain control statements and function calls. The main role of the control part is to execute test cases. The control part is not allowed to directly communicate with the SUT, to set a verdict, or to create dynamic configurations. These operations must be performed only within test cases.

Preprocessing macros are used in definition or control parts to locate the position of the macro call. TTCN-3 compiler replaces these macros with their **charstring** or **integer** values. More precisely, these values are inserted in the program source code instead of the macro calls. By the convention, the macro's names are enclosed by underscores. Currently, TTCN-3 offers the following preprocessing macros: _MODULE_, _FILE_, _BFILE_, _LINE_, and _SCOPE_.

The value of the macro _MODULE_ is the name of TTCN-3 module in which the macro was called.

The value of the macro _FILE_ is the full pathname (ending with the basic file name) of the file in which the macro was called.

The value of the macro _BFILE_ is the basic file name (without its path) of the file in which the macro was called.

The value of the macro _LINE_ is the number of the source code (i.e., file) line in which the macro was called.

The value of the macro _SCOPE_ depends on whether the corresponding scope is named or unnamed. The following basic scopes are named: the module, the control part (has a special name "Control"), the function, the component, the test case, the altstep, the template, and the user-defined type. If the corresponding scope is named, the value of the macro _SCOPE_ is its name; otherwise, the value is the name of the next higher basic scope.

3.9.3 Single Component TTCN-3 Test Suites

Although TTCN-3 resembles a common programming language, it's a domain-specific language for developing test cases, which defines the interaction between the test system and the SUT. In this section, we study the message-based communication with the SUT and test cases executed on a single test component (i.e., nonconcurrent TTCN-3 test suites).

We introduce the concepts for message-based communication and single component test suites through examples for testing the Address Resolution Protocol (ARP) server. So, the test setup is such that test cases executing on a test system (also called the tester) imitate an ARP client, whereas the SUT is the real ARP server under testing, see Figure 3.45.

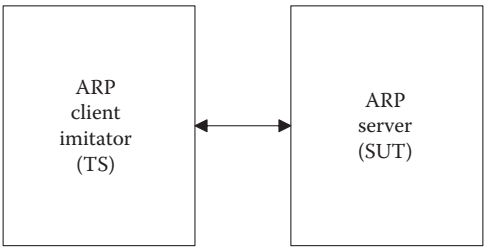


FIGURE 3.45
Test configuration for testing the ARP Server.

The main task of the ARP is to map a given network address, such as the Internet Protocol version 4 (IPv4) address, into the corresponding physical (or hardware) address, such as Ethernet address, which is also known as the Media Access Control (MAC) address. The ARP is a simple client–server protocol, which uses a simple message format containing one address resolution request or response. The size of the ARP messages depends on the size of the particular network and physical addresses. For example, the size of the IPv4 address is 32 bits (4 bytes), the size of the MAC address is 48 bits (6 bytes), and the size of ARP messages used to map IPv4 to the MAC addresses is 28 bytes.

The fields of the ARP message used for mapping IPv4 to MAC addresses are as follows (we refer to individual bytes, also called octets, of the message by using their index, which starts from 0):

- Hardware type (HTYPE), bytes 0–1, specifies the type of the physical address (for Ethernet, HTYPE is equal to 1).
- Protocol type (PTYPE), bytes 2–3, specifies the network protocol (for IPv4, PTYPE is equal to 0x0800).
- Hardware address length (HLEN), byte 4, is the length of the hardware address (for Ethernet, HLEN is equal to 6).
- Protocol address length (PLEN), byte 5, is the length of the network address (for IPv4, PLEN is equal to 4).
- Operation (OPER), bytes 6–7, specifies the operation that the sender is performing (1 for request, 2 for reply).
- Sender hardware address (SHA), bytes 8–13, is the sender's MAC. In the message ARP request, this field is the MAC of the host sending the request. In the message ARP reply, this field is the MAC of the host that the request was looking for, i.e., the result of the request mapping.
- Sender protocol address (SPA), bytes 14–17, is the sender's IPv4 address.

- Target hardware address (THA), bytes 18–23, is the receiver's MAC. In the message ARP request this field is ignored. In the message ARP reply, this field is the MAC of the host that sent the initial message ARP request.
- Target protocol address (TPA), bytes 24–27, is the receiver's IPv4 address.

We may describe the types of the fields of the ARP message by the following supplementary types (note that generally we may specify hexadecimal numbers using the construct '*h_num*'H, where *h_num* is a hexadecimal number):

```
type integer Int8 (0..'FF'H)
type integer Int16 (0..'FFFF'H)
```

where *Int8* corresponds to a single byte field and *Int16* corresponds to a double byte field. Then we may define possible values of the field OPER using the following enumerated type (note that generally we may explicitly assign a value to an enumeration element by writing the particular value enclosed in the parenthesis after the particular enumeration element name):

```
type enumerated ARPOperation (
  e_ARPRequest(1),
  e_ARPReplay(2)
);
```

Using these supplementary types, we may describe the ARP message by the following record type:

```
type record ARPMessage {
  Int16 htype,
  Int16 ptype,
  Int8 hlen,
  Int8 plen,
  Int16 oper,
  charstring sha,
  charstring spa,
  charstring tha,
  charstring tpa
}
```

Finally, we may construct individual ARP messages using the following parametrized template:

```
template ARPMessage t_ARPMessage(
  Int16 p_oper, Int48 p_sha, Int32 p_spa, Int48 p_tha, Int32 p_tpa
```

```
):= {
  htype := 1,
  ptype := 0x0800,
  hlen := 6,
  plen := 4,
  oper := p_oper,
  sha := p_sha,
  spa := p_spa,
  tha := p_tha,
  tpa := p_tpa
}
```

The ARP operates as follows: Assume that the router R has to deliver an IPv4 datagram to the host H, which for example has the IPv4 address 192.168.0.48 and the MAC address 00:EB:24:B2:05:C8. First, R will have a look in its own local routing table for the entry corresponding to H's IPv4 address. If R finds it there, then R reads the H's MAC address from that entry and uses it to perform direct datagram delivery to H.

If R does not find the entry for the IPv4 address 192.168.0.48, then R broadcasts the message ARP request for this IPv4 address by sending the Ethernet frame to the destination MAC address FF:FF:FF:FF:FF:FF. The ARP server S receives this message, finds the mapping in its local table, creates the corresponding message ARP reply, and sends it to R, which, in turn, performs direct datagram delivery to H, and updates its local routing table accordingly.

The tester (i.e., test system) may test ARP by executing a simple test case, which first sends the message ARP request (with SPA set to its IPv4 address, SHA set to its MAC address, and TPA set to the IPv4 address 192.168.0.48; THA is ignored), and then receives the message ARP reply with the required mapping (with SPA set to the IPv4 address 192.168.0.48 and SHA set to the MAC address 00:EB:24:B2:05:C8), see the MSC in the Figure 3.46. If the

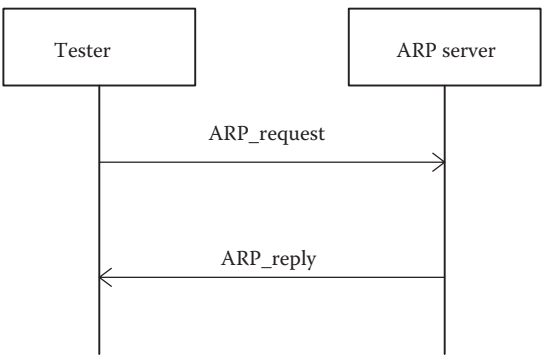


FIGURE 3.46
MSC for mapping the IPv4 address into the corresponding MAC address.

received message ARP reply contains the correct mapping, the tester would set the test verdict to **pass**; otherwise, it would set the test verdict to **fail**.

Here, we introduce the following concepts for message-based communication and single component test suites:

- Ports
- Components
- Test Cases
- Templates
- Message-Based Communication
- Timers
- Alt Statement
- Altsteps
- Default Altsteps
- Functions

Ports are used for exchanging messages. The messages sent to a port are immediately delivered to the related receiver, whereas the messages received from a port are stored in the unbounded FIFO queue, which is implicitly assigned to a port. Although the queue is theoretically unbounded, i.e., of infinite length, TTCN-3 implementations may introduce some practical limits.

Directions of messages exchanged over ports are defined from the test system point of view. There are the three possible message transfer modes for exchanging messages over ports, namely the mode **out**, the mode **in**, and the mode **inout**. The mode **out** is used for sending messages from the test system to the SUT, the mode **in** is used for receiving messages sent from the SUT to the test system, whereas the mode **inout** is used for the bidirectional exchange of messages between the test system and the SUT.

Generally, a single port may be used for exchanging more message types. Moreover, messages of different message types may be exchanged over the same port in the same or in the different message transfer modes. In the most general example, messages of the types *A*, *B*, and *C* may be exchanged over the same port in the transfer modes **out**, **in**, and **inout**, respectively.

Most frequently we will use a single port for the exchange of a single type of message in a single transfer mode. For example, we may define the message port type *ARPPort* for the bidirectional exchange of messages, or the type *ARPMessage* in the message transfer mode **inout**, in order to test the target ARP server:

```
type port ARPPort message {
  inout ARPMessage
};
```

However, sometimes we will need to define different message types to be exchanged over the same port type in various message transfer modes. For example, imagine that we want to test the Email server. Since Email clients use SMTP protocol for sending email messages to the Email server, and POP3 protocol for receiving email messages from the Email server, we would define one message port type with two different message types, for example, as follows:

```
type port MailPort message {
  inout SMTPMessage;
  inout POP3Message
}
```

Components are used for executing test cases. A component may have its local state that comprises its constants, variables, and timers. The component's interface is defined by its ports. In order to define a component type, we have to provide the list of particular port instances used by that component type, where each item in that list indicates the type of the port and the name of the port instance. It is not necessary that all the ports have different types. Some of the ports may have the same type, but their names must be different, i.e., unique.

For example, we may define the component type *ARPTester*, which uses a single port instance of the type *ARPPort* with the name *serverPort*, as follows:

```
type component ARPTester {
  port ARPPort serverPort
}
```

Optionally, we may define component's constants, variables, and timers, within the component's type definition. As shown in the previous section, constants are defined by their type, name, and value; variables are defined by their type and name and optional initial value; and timers are defined by their name and optional default duration of the type **float**. It is important to notice that each instance of a component type has its own instances of the ports, variables, and timers (i.e., they are analogous to nonstatic class attributes in programming languages).

For example, we may extend the previous definition of the component type *ARPTesterS* by introducing the constant *c_maxRequests* (the max number of ARP requests that an *ARPTester* may send in a burst, i.e., without waiting for a reply before issuing the next request), the variable *v_noRequests* (the number of requests sent to the SUT), and the timer *t_inactive* (that may bound the time interval for waiting the reply from the SUT), with the default duration of 0.5 s. The extended type *ARPTesterS* is as follows:

```
type component ARPTesterS {
  const integer c_maxRequests := 1000;
  var integer v_noRequests;
  timer t_inactive := 0.5;
  port ARPPort serverPort
}
```

Test cases are used to describe the expected behavior of the SUT, and to set the test verdict depending on the real behavior of the SUT. More precisely, test cases define the behavior of the main test component within a given test configuration that may generally have more test components. As its name suggests, a single component TTCN-3 test suite's test configuration has a single test component, which must be the main test component.

The Test System Interface (TSI) is the interface between the TS and the SUT. In case of a single component TTCN-3 test suite, TSI is completely defined by the set of ports of the main test component, thus TSI is defined implicitly and there is no need to define it separately.

When writing a test case, we use the clause **runs on** to specify the component type that will execute that test case. Most frequently, a test case will not have parameters, and in such a case we simply omit the list of formal parameters by writing the empty pair of parenthesis after the test case name. For example, the following empty test case *tc_nop* (which does not perform any operation) is designed to be executed on the component type *ARPTester*, which has no parameters:

```
testcase tc_nop() runs on ARPTester {};
```

We have already introduced possible test verdicts (**none**, **pass**, **inconc**, **fail**, and **error**) without going too much into detail. Actually, each test component has its own local verdict, which is a variable of the type **verdicttype** that we may set or get using the test component's operations **setverdict** or **getverdict**, respectively. The exception is the verdict **error**, which can be set only by the runtime execution system (within the error handling routine) and cannot be set by a test case. Like any other variable, we may log the local verdict current value by the statement **log**.

As already mentioned, the initial value of the local verdict (i.e., its default value) is the verdict **none**. For example, since the test case *tc_nop* performs no operation, its final verdict is the verdict **none**, too.

Unlike simple variables, the possible values of the local verdict are not just elements of a conventional enumeration. Instead, verdicts are assigned different strengths, such that all the verdicts are ordered by their strength, from the weakest (**none**) to the strongest (**error**), according to the following list: <**none**, **pass**, **inconc**, **fail**, **error**>. Assignment of a value to the local verdict is governed by the following important rule: The current value of the local verdict can be assigned the next value only when the next value is stronger than the current value.

For example, the value of the local verdict can be changed from **none** to **pass** or **fail**, but it cannot be changed, for example, from **fail** to **pass**. Therefore, in the test case *tc_remains_fail*, as shown below, the final test verdict remains **fail**, because the assignment of the verdict **pass** after the assignment of the verdict **fail** is not possible (and thus the runtime execution system just ignores it):

```
testcase tc_remains_fail() runs on ARPTester {
  var verdicttype current_verdict;
  setverdict(fail);
  …
  // later in the code…
  setverdict(pass);
  current_verdict = getverdict; // verdict remains fail
};
```

Besides the local verdict, a test component also has the implicit variable of the type **charstring**, which may be used to describe the reason for the particular verdict assignment. This variable is assigned by the **setverdict** operation and the reason string is passed as one or more optional parameters at the end of the **setverdict** operation's parameter list (these parameters are specified the same way as those for the **log** statement). For example, in the test case *tc_always_pass*, we describe the reason for setting the test verdict to **pass**:

```
testcase tc_always_pass() runs on ARPTester {
  // Check the SUT behavior
  setverdict(pass, 'The SUT behavior was as expected.')
};
```

We should note that in the case of a single component TTCN-3 test suite, the overall test verdict is equal to the local verdict of the main test component (whereas in the case of the multi component test suite, it is evaluated based on the local verdicts of individual test components).

As we have already seen, a test case is executed from the control part by the statement **execute**, which returns the overall test case's verdict. The verdict returned by the statement may be stored in the variable of the type **verdicttype** for further processing, or it may be ignored if it is not needed. Note that assignments to the user-defined variable of the type **verdicttype** are not governed by the assignment rule for the test component's local verdict, because it is a simple variable, so its value can be changed freely.

The second parameter of the operation execute is optional, and when it is supplied it defines the upper bound on the test case execution time. Under the hood, the runtime execution system starts the corresponding timer, and if the timer expires it terminates the test case with the verdict **error**. We should

note that even if some of the test cases have the overall test verdict **error**, other test cases defined within the control part will be executed as requested.

The following example illustrates the control part that executes the three previously introduced test cases. The execution time for all the test cases is bounded to the time interval of 5s and the return verdict is stored into the user-defined variable *result* for all the executions:

```
control {
  var verdicttype result;
  result := execute(tc_nop(), 5.0);
  result := execute(tc_remains_fail(), 5.0);
  result := execute(tc_always_pass(), 5.0);
};
```

Usually, a control part, such as the one shown above, is just a list of execute statements, but when needed, we may use conditional statements and loops (introduced in the previous section) within a more complex control part.

Like functions, test cases may have **in**, **out**, and **inout** parameters. The **in** parameters are passed by a value, whereas the **out** and **inout** parameters are passed by a reference. In the latter case, changes of parameters within the test case cause updates of real parameters in the control part. But, if the test case verdict is **error**, the values of **out** parameters are undefined.

The function's restrictions of its real parameters (which we have already seen) apply to test case parameters, too. The real **inout** parameter cannot be uninitialized, and the real **out**, as well as the real **inout** parameter, cannot be a constant expression. For example, the following test case *tc_counting* has the **inout** parameter *p_count*, which may be used for counting the number of test case executions:

```
testcase tc_counting(inout p_count) runs on ARPTester {
  p_count := p_count + 1;
  setverdict(pass);
};
```

Within the control part, we may define the initialized **inout** variable *v_count* in order to count the number of test case executions:

```
control {
  var integer v_count := 1;
  // execute tc_counting 10 times
  for ( integer i := 0; i < 10; i := i + 1 ) {
  log("v_count = ", v_count);
  execute( tc_counting(v_count) );
  }
}
```

The local constants, variables, and timers of the test component which the test case **runs on**, are in the scope of this test case. These constants, variables, and timers may be used the same way as ordinary test case's local variables. This concept is similar to the concept of inheriting attributes of a supper class in a subclass in programming languages.

In TTCN-3, a function may also inherit local constants, variables, and timers of the test component on which it runs. Note that for all the test cases and functions running on the same test component, these inherited local constants, variables, and timers appear as global entities, and we should use them carefully (the same way we use global variables in other programming languages).

In the following test case *tc_using_comp_vars*, we set the local variable of test component *ARPTesterS* the same way as we set the test case's local variable *v_current*:

```
testcase tc_using_comp_vars() runs on ARPTesterS {
 var integer v_current := 1
 …
 v_noRequests := 10;
 v_current := 1;
 …
}
```

A test case implicitly terminates with its last statement. We may explicitly terminate a test case using the operation **stop** or the operation **testcase.stop**. We use the operation **stop** to terminate an error-free test case execution and the operation **testcase.stop** to terminate an erroneous test case execution. The operation **stop** returns the overall test verdict to the control part (analogously to the statement **return** that returns the return value of the called function to the calling function in other programming languages). On the other hand, the operation **testcase.stop** sets the test verdict to **error** and terminates the test case. We may use the operation's optional arguments to indicate the reason for termination (the same way as we use the optional arguments of the operation **setverdict**).

Templates are used to define messages exchanged between the test system and the SUT. When we want to send a particular message from the test system to the SUT, we use the template instance that defines a single value of the corresponding type, i.e., that particular message. But, when we want to receive a reply from the SUT, we would more frequently use matching expressions with template instances specifying more possible reply messages.

Generally, a template defines a set of values of a given type. This set may contain just a single value, more values, or even all the values of the given type (we specify all the values using the character '?'). In the example below, we use the nonparametrized template *t_fixedARPRequest* to define the fixed ARP request message from the test system to the SUT, with SPA set to "192.168.0.40" (this is the test system's IPv4 address), SHA set to "00:EB:24:B2:05:C0" (this is

the test system's MAC address), TPA set to "192.168.0.48" (this is the IPv4 address that has to be resolved), and THA set to 0 (actually, it could be any value, because ARP protocol ignores THA field in the ARP request message):

```
template ARPMessage t_fixedARPRequest () := {
  htype := 1,
  ptype := 0x0800,
  hlen := 6,
  plen := 4,
  oper := 1,
  sha := "00:EB:24:B2:05:C0",
  spa := "192.168.0.40",
  tha := 0,
  tpa := "192.168.0.48"
}
```

On the other hand, the previously introduced parametrized template *t_ARPMessage* defines a subset of all the possible values of the record type *ARPMessage*, with the first four fields fixed to the values 1, 0x0800, 6, and 4, respectively.

Although templates are used to specify values, they are not values. Even a single-valued template is not a value. Thus, a template cannot be directly used in an expression.

However, templates can be passed as **in** parameters to functions and test cases. Such a parameter must be defined with the additional keyword **template** in order to distinguish it from the simple value. For example, the following test case has the template as its input parameter:

```
testcase tc_withParam(
 in template t_ARPMessage p_msg
) runs on ARPTester {
  // some statements that depend on p_msg
};
```

Message-based communication between the test system and the SUT is conducted over TSI ports in order to effectively test the SUT. The type **port** supports the three main operations, namely **send**, **receive**, and **check**. The operation **send** sends the specified message to the SUT. The operation **receive** compares the received message with the specified template, and if they match, it receives the message from the port's queue; otherwise it blocks. The operation **check** is similar to the operation **receive**, but it does not remove the received message from the port's queue. Besides receiving a message from a single port, it is also possible to receive a message from **any port**. In the following paragraphs, we study these operations in more detail.

The port's operation **send** sends the particular message (single value template instance) over the specified port. For example, the following test case creates the ARP request message *req_msg*, with SPA set to "192.168.0.40" (the test system's IPv4 address), SHA set to "00:EB:24:B2:05:C0" (the test system's MAC address), TPA set to "192.168.0.48" (the IPv4 address that has to be resolved), and THA set to 0 (actually, it could be any value), and sends this message over the port *serverPort* to the SUT:

```
testcase tc_resolve_part_1() runs on ARPTester {
  // create the ARP request message
  ARPMessage req_msg := t_ARPMessage(
    1,                          // ARP operation: 1 – request
    "00:EB:24:B2:05:C0",        // test system's MAC address
    "192.168.0.40",             // test system's IPv4 address
    0,                          // this field is ignored by ARP
   "192.168.0.48"               // target IPv4 address to be resolved
 );
  // send the ARP request message
  serverPort.send(req_msg);

  // part 2 - to be finished later
};
```

The state of the SUT cannot influence the execution of the operation **send**, which is executed by the test system. Once the message is delivered over the specified port, the operation **send** is finished, and the test case proceeds to the next statement following it, no matter whether SUT really received the message or not.

When we define a template using a simple type rather than a record, the particular template instance might not be distinguished from the ordinary value of the corresponding type. In such a case, the value must be preceded by a type name. For example, assume that we defined the template *t_MyIPAddresses* using the type charstring, and assume that "128.0.0.0" is a member of *t_MyIPAddresses*. In order to **send** the value "128.0.0.0" as one of the *t_MyIPAddresses* instances, we must explicitly write the template name before the particular value:

```
type charstring t_MyIPAddresses {"128.0.0.0", …};
somePort.send(t_MyIPAddresses: "128.0.0.0");
```

The port's operation **receive** is generally used for receiving messages from the SUT. Unlike the operation **send**, its argument is a template that may specify more possible SUT replies rather than just one particular SUT reply (which is allowed as a special case). Also, the operation **receive** is a blocking operation, whereas the operation **send** is a nonblocking operation.

The operation **receive** performs two steps. In the first step, it compares the message at the head of the port's queue with the specified template. If this message is a member of the set of messages specified by the template, we say that the message matches the template. More precisely, if the template specifies a single message, the message at the head of the queue must be that message. If the template specifies a subset of messages of the message type that may be received over the specified port, the message at the head of the queue must be a member of that subset. Finally, if the template specifies any message of the corresponding message type, then the message at the head of the queue must be of that type.

In the second step of the operation **receive** there are two possible cases. If the message at the head of the queue matches the template, this message is dequeued from the head of the queue and delivered to the receiving process, which proceeds to the next statement that follows the operation **receive**. If the message at the head of the queue does not match the template, and if there are no alternatives, then the receiving process blocks within the operation **receive** (we introduce alternatives later in the following text).

The message at the head of the queue may mismatch the template in two possible cases. The first case is when the queue is empty. No message mismatches any template, and consequently the receiving process blocks. The second case is when there is some message at the head of the queue that mismatches the template, so the receiving process again blocks. However, there is a fundamental difference between these two cases. In the latter case, the receiving process blocks forever (even if the right message is received later, because it will still not be positioned at the head of the queue), whereas in the former case, the receiving process blocks temporarily. If the right message is received later, the receiving process would be unblocked.

The following test case *tc_resolve* tests the whole ARP. Its first part is the same as in the previous test case *tc_resolve_part_1*. In its second part, the test case *tc_resolve* creates the expected ARP reply message *rpy_msg*, with SPA set to"192.168.0.48" (the IPv4 address that has to be resolved); SHA set to "00:EB:24:B2:05:C8" (the expected MAC address that should be the result of the ARP resolution); TPA set to"192.168.0.40" (test system's IPv4 address); and THA set to "00:EB:24:B2:05:C0" (test system's MAC address), which receives this message over the port *serverPort*, and sets the test verdict to **pass**.

```
testcase tc_resolve() runs on ARPTester {
 // create the ARP request message
 ARPMessage req_msg := t_ARPMessage(
  1,                          // ARP operation: 1 - request
  "00:EB:24:B2:05:C0",        // test system's MAC address
  "192.168.0.40",             // test system's IPv4 address
  0,                          // this field is ignored by ARP
  "192.168.0.48"              // target IPv4 address to be resolved
 );
  // send the ARP request message
```

```
  serverPort.send(req_msg);

  // part 2 – create ARP reply, receive it, and set test verdict
  // create the ARP reply message
 ARPMessage rpy_msg := t_ARPMessage(
  2,                          // ARP operation: 2 - reply
  "00:EB:24:B2:05:C8",        // target MAC address – expected value
  "192.168.0.48",             // target IPv4 address to be resolved
  "00:EB:24:B2:05:C0",        // test system's MAC address
  "192.168.0.40"              // test system's IPv4 address
 );
  // receive the ARP reply message
  serverPort.receive(rpy_msg);
  // set test verdict to pass
  setverdict(pass);
};
```

In the previous test case, the operation **receive** may block the receiving process temporarily if the test system still did not receive a reply from the SUT. Alternatively, the operation receive may block forever if the test system received the message that mismatched the expected message *rpy_msg*. The receiving process will not block, or will be unblocked, if the test system receives the expected message *rpy_msg*. Once this expected message is received, the test case will set the test verdict to **pass** and it will successfully terminate.

The operation **receive** also offers an option to save the received message into the specified variable of the corresponding type (e.g., the type that is used in the definition of the template). The syntax of the statement using this option is as follows:

port.**receive**(*template*) **-> value** *variable*

where *port* is the name of the port over which the message is to be received, *template* is the name of the template that the received message should match, and *variable* is the name of the variable where the received message should be stored.

Alternatively, by using the operation **receive** without the argument, we may receive any message over the specified port, but we cannot save that message. Of course, the received message must be of the correct type. For example, the following statement will receive any message of the type *ARPMessage*:

serverPort.**receive**;

Like in the case of the operation **send**, if the type of the operation's argument could not be uniquely determined, it must be specified explicitly as follows:

port.**receive**(*type: template*)

where *port* is the name of the receiving port, *type* is the name of the message type, and the *template* is the name of the template.

The port's operation **check** receives the message from the specified port, but it does not remove it from the port's queue. The receiving process will block if the queue is empty or if the message at the head of the queue mismatches the specified template. Alternatively, if the message at the head of the queue matches the specified template, the operation check successfully finishes, and the receiving process proceeds to the next statement following it.

The operation **check** is the operation on the specified port whose argument is the operation **receive** with its argument. For example, the following statement checks any message on the port *serverPort*:

serverPort.**check**(**receive**);

Alternatively, the following statement checks the particular ARP reply *rpy_msg* on the port *serverPort*:

serverPort.**check**(**receive**(*rpy_msg*));

Like the operation **receive**, the operation **check** offers the option for saving the checked message into the specified variable. The syntax of the statement for using this option is the same as for the operation **receive**. For example, the following statement saves the checked message *rpy_msg* into the variable *v_msg* of the type *ARPMessage*:

ARPMessage v_msg;
serverPort.**check**(**receive**(*rpy_msg*)) -> **value** *v_msg*;

Besides receiving and checking messages on the particular port, we may receive or check messages on **any port**. We may want to do this in order to receive or check the unexpected messages and we may do this simply by using the keyword **any port** as the port name in the corresponding statements. Of course, sending some message on any port would be an ambiguous operation, thus this option is not supported.

For the sake of illustration, assume that *SysTester* is the test component with two ports, namely *serverPort* and *serverPort2*. Further assume that the port *serverPort* connects the test system with the primary ARP server, and the port *serverPort2* connects the test system with the secondary ARP server (which is a backup in case of the primary ARP server failure).

Generally, we may receive any message on **any port** by using the operation receive on **any port** and without a template, as follows:

any port.**receive**;

If this statement is executed on the test component *SysTester*, it would block until there is a message in at least one of the two message queues.

Alternatively, if both queues contain messages, this statement would randomly select one of the two queues, and it would dequeue the message from the head of the selected queue. However, in this case, there are no means to determine from which port the message was dequeued.

Alternatively, we may receive the specified message(s) on any port. For example, if the following statement is executed on the test component *SysTester*, it would receive the message *rpy_msg* either from the port *serverPort* or the port *serverPort2*:

any port.**receive**(*rpy_msg*);

Again, it would not be possible to determine whether the message *rpy_msg* was received from the port *serverPort* or the port *serverPort2*. In this particular example, this would mean that the system as a whole (primary plus secondary ARP servers) reacted as expected. However, in some other protocols, receiving excepted messages from **any port** might not be what we are really looking for. Receiving unexpected messages from any port is the intended usage of the keyword **any port**.

Like in the case of the ordinary receipt of the specified port, we may save the message received on any port into the specified variable. The syntax is the same. For example, if the following statement is executed on the test component *SysTester*, it would save the received message *rpy_msg* (received from either of two available ports) into the variable *v_msg*:

any port.**receive**(*rpy_msg*) -> **value** *v_msg*;

Timers are used to describe the protocol's timing properties. The moment in time is represented by the nonnegative floating point number (**float**). The type **timer** supports the five main operations: **start**, **stop**, **timeout**, **read**, and **running**. The operation **start** starts the specified timer, the operation **stop** stops the specified timer, the operation **timeout** waits for the specified timer to expire, the operation **read** returns the duration since the specified timer was started, and the operation **running** returns the Boolean indicator indicating whether the specified timer is running (the indicator has the value **true** if the timer running; otherwise it has the value **false**).

We may declare a timer within the test component, the test case, the control part of a module, the function, or the altstep. Each timer exists only within the scope in which it was declared. Once the timer's scope is left, the timer is destroyed, and thereafter becomes unavailable. We may declare a timer without explicitly specifying its default duration. For example, the following declaration declares the timer *t_T1* without the explicit default duration:

timer *t_T1*;

Alternatively, we may declare a timer with the explicit default duration. For example, the following declaration declares the timer *t_T2* with the default duration of 1s:

```
timer t_T2 := 1.0;
```

We start the specified timer by the operation **start**, which has the timer duration as an optional argument. If we use this optional argument, and if the timer was declared with the explicit default duration, the value of the optional argument will overwrite the default value. For example, the following statement starts the timer *t_T2* for the duration of 2s:

```
t_T2.start(2.0);
```

We typically use the operation **timeout** to simulate the desired rhythm of messages that are sent towards the SUT. For example, imagine that we want to send the ten *req_msg* messages towards the SUT over the port *serverPort*, with the 1s time interval between two adjacent messages. We may do this by the following snippet of code:

```
for ( integer i := 0; i < 10; i := i + 1 ) {
  serverPort.send(req_msg);
  t_T2.start;
  t_T2.timeout;
}
```

We may stop the running timer by the operation **stop**. It is important to remember that the timer's states stopped and expired are two different states. Note that the operation **timeout** on the previously stopped timer would block forever, because this timer would remain in the state stopped and would never go (back) to the state expired. Another important detail to remember is that starting the running, or expired, timer is equivalent to first stopping and then restarting the timer.

The operation **running** and the operation **read** are typically combined. In the following example, we start the timer *t_T1* with the duration of 10s and then while it is running, we use the timer *t_T2* to report, every 1s, the time that elapsed from the moment when the timer *t_T1* was started:

```
t_T1.start(10.0);
while(t_T1.running) {
  log(t_T1.read, " seconds elapsed since t_T1 was started…");
  t_T2.start(1.0);
  t_T2.timeout;
}
log("t_T1 expired.");
```

We may pass timers as **inout** arguments to altsteps or functions, but we cannot pass them to test cases. A timer does not need to be in the running state in order to be passed as an argument.

Alt Statements are used to combine several blocking operations as possible alternatives to continue process execution, in order to avoid unbounded blocking of individual blocking operations. The statement **alt** executes the first blocking operation that is ready to proceed.

For example, as already mentioned, the standalone operation **receive** will block forever if no message, or some unexpected message, is received. The usual way to overcome this situation is to guard this blocking operation **receive** by using a timer. We do this by starting a timer and using the **alt** statement with two alternatives, namely the operation **receive** on the specified port, with the template specifying the expected message(s), and the operation **timeout** on the running timer.

However, this solution with these two alternatives does not eliminate possible indefinite blocking in case when some unexpected message is received on the specified port. Therefore, if we want to completely eliminate indefinite blocking we must use the statement **alt** with the three alternatives in the order listed below:

- The operation **receive** on the specified port with the template specifying the expected message(s)
- The operation **receive** on the specified port without any template, which is used to receive the unexpected messages.
- The operation **timeout** on the running timer, which is used to terminate indefinite blocking in case when no messages are received in some reasonable interval of time (which is equal to the duration of the timer)

This order of alternatives in the statement **alt** is important, because the alternatives are evaluated from top to bottom, and the first one that is ready to proceed will be executed. So, the position of the alternative may be seen as its priority, because if two alternatives are ready to proceed, the one that is closer to the top of the list of alternatives will get executed.

So, how should we order the alternatives? Generally, we put the alternatives for the expected messages on the top of the list, and then we proceed to various kinds of unexpected messages and errors going down the list.

The following test case uses this strategy to test the ARP:

```
testcase tc_resolve_guarded() runs on ARPTester {
 timer t_T1;
 // create the ARP request message
 ARPMessage req_msg := t_ARPMessage(
  1, "00:EB:24:B2:05:C0", "192.168.0.40", 0, "192.168.0.48"
```

```
  );
  // send the ARP request message
  serverPort.send(req_msg);
  // part 2
  // create the ARP reply message
  ARPMessage rpy_msg := t_ARPMessage(
    2, "00:EB:24:B2:05:C8", "192.168.0.48",
    "00:EB:24:B2:05:C0", "192.168.0.40"
  );
  // start the timer t_T1 with duration 1s
  t_T1.start(1.0);
  // use the statement alt with 3 alternatives
  alt {
    []serverPort.receive(rpy_msg) { // rpy_msg received
      t_T1.stop;
        setverdict(pass);
      };
    []serverPort.receive { // unexpected message received
      t_T1.stop;
        setverdict(fail);
      };
    []t_T1.timeout { // timer expired
      setverdict(fail)
      }
  }
};
```

What happens if a message arrives on some port, or some timer expires, while the other alternative is evaluated? Obviously, immediate and continuous reevaluation of all the alternatives would lead to race conditions. Therefore, the statement **alt** uses the concept of the **snapshot** in order to keep the top-down order of evaluation and to avoid race conditions. More precisely, the statement **alt** performs the following steps in a loop until it breaks from it:

- Take a snapshot of the current state of the test component.
- Evaluate all the alternatives from the top to the bottom of the list.
- When the first alternative that is ready to proceed is found, break this loop and execute that alternative.

Furthermore, the statement **alt** offers the option to specify **Boolean guards** for its alternatives, which we did not use so far. Actually, the empty square brackets that we used to mark the beginning of an alternative are the placeholder for an optional Boolean guard. The Boolean guard is the Boolean expression, which evaluates to the values **true** or **false**.

The statement **alt** considers only the alternatives whose Boolean guards evaluate the value **true**, and skips the alternatives whose Boolean guards evaluate the value **false**. The special guard **else** is used to mark the default alternative at the end of the list of alternatives, which will be selected if none of the previous alternatives were selected.

In the following example, we use two Boolean guards to guard the reception of the corresponding messages, and we also use the default guard **else**:

```
alt {
  [select_msg == 1] pt.receive(t_msg1) { setverdict(pass); };
  [select_msg == 2] pt.receive(t_msg2) { setverdict(pass); };
  [else] { setverdict(fail); }
}
```

In the previous example, the test verdict would be set to **pass** if the first Boolean guard evaluates the value **true** and the message *t_msg1* is received over the port *pt*, or if the second Boolean guard evaluates to the value **true** and the message *t_msg2* is received over the port *pt*. Otherwise, the test verdict would be set to **fail**.

Generally, the Boolean guards in the list of alternatives do not have to be orthogonal and complete, i.e., more or none of them may evaluate the value **true**. If more Boolean guards evaluate the value true, the corresponding alternatives are evaluated top-down until the first alternative ready to proceed is selected. If none of the Boolean guards evaluate the value **true** and we do not use the default guard **else**, there are two possible cases: (1) the Boolean guards are independent of the snapshot and (2) the Boolean guards are dependent on the snapshot.

If the first case, the statement **alt** would block forever, which is considered to be a test case design error. In the second case, there is a chance that the statement **alt** will not block forever, because it will continue taking snapshots in a loop, and for some future snapshot some Boolean guards may evaluate the value true. However, there is the risk that this does not happen, because of a design error, so we would be better off by avoiding such designs.

Motivated by these concerns, TTCN-3 standard forbids using operations in Boolean guards whose results may change in repeated evaluations, such as checking whether a timer is running or not. Also, functions that are called from Boolean guards must not change the current snapshot. The examples of forbidden operations, within such functions, are the operation **receive** on a port; the operations **start**, **stop**, and **timeout** on a timer; and operations that update the test component's local variables.

As discussed so far, the statement **alt** may be seen as a selection of alternatives—once the alternative with the highest priority that is ready to proceed is selected, it is executed, and the execution continues with the next statement following the statement **alt**. But, sometimes we would like to repeat the whole selection from the beginning. A traditional way to do it

would be to introduce a loop with a break indicator around the statement **alt**. The more elegant way to do it is to use the statement **repeat**.

The statement **repeat** repeats the whole enclosing statement **alt** from the very beginning—the Boolean guards and the alternatives are evaluated again and the next alternative is selected. We may use the statement **repeat** only within the alternatives of the statement **alt** (typically, as the last statement in the alternative) or within the alternatives of an **altstep**. The way the statement **repeat** operates is somewhat similar to the tail recursion in functional programming languages.

As an example, we may use the statement **repeat** to construct a simple ARP server robustness test. Sometimes, the SUT may return the correct reply to the single request, but when the same request is repeated more times, the SUT may become overloaded or some internal synchronization error may lead to a failure, which may cause incorrect replies from the SUT or absence of replies. To test robustness of the SUT, we adapt the test case *tc_resolve_guarded* such that we send the burst of the same ten ARP requests (by using a simple for loop) and then we expect to receive the same ten ARP replies (by using the statement **repeat**):

```
testcase tc_resolve_robustness() runs on ARPTester{
 timer t_T10;
 // create the ARP request message
 ARPMessage req_msg := t_ARPMessage(
  1, "00:EB:24:B2:05:C0", "192.168.0.40", 0, "192.168.0.48"
 );
 // send the burst of 10 ARP request messages
 for( integer i := 0; i < 10; i := i + 1 ){
  serverPort.send(req_msg);
 }
 // part 2
 // create the ARP reply message
 ARPMessage rpy_msg := t_ARPMessage(
  2,"00:EB:24:B2:05:C8","192.168.0.48",
  "00:EB:24:B2:05:C0","192.168.0.40"
 );
 // start the timer t_T1 with duration 10s
 t_T10.start(10.0);
 // use the statement alt and repeat to receive 10 ARP replies
 alt{
  [] serverPort.receive(rpy_msg){ // rpy_msg received
     setverdict(pass);
     repeat; // repeat in order to receive the next reply
   };
  [] serverPort.receive{ // unexpected message received
     t_T10.stop;
```

```
    setverdict(fail);
  };
 [] t_T10.timeout{ // timer expired
    setverdict(fail)
  }
}
};
```

So far, we have seen only the statements **alt** with more alternative blocking operations. Since the statement **alt** with a single alternative behaves as a single alternative without the enclosing statement **alt**, we would naturally write the single alternative as a stand-alone operation (without the enclosing statement **alt**).

Interestingly enough, and for the reason that would become apparent later on when we introduce **default altsteps**, according to the TTCN-3 standard, a stand-alone blocking operation will be treated by implicitly wrapping it into the enclosing statement **alt**. For example, the stand-alone blocking statement:

```
serverPort.receive(rpy_msg);
```

is implicitly expanded to:

```
alt {
 [] serverPort.receive(rpy_msg) {}
}
```

Altsteps are named groups of alternatives, which may be referred to within the statement **alt**. Like functions, they may have parameters, but unlike functions they may use the Boolean guards and the operations **receive** and **timeout**. The following typical altstep has the timer *p_timer* as its parameter, and it checks the timeout condition on this timer:

```
altstep alt_timeout(inout timer p_timer) {
 [] p_timer.timeout { setverdict(fail) }
};
```

We may now use the altstep *alt_timeout* within the statement **alt**, for example, in order to bound time interval for waiting the message *rpy_msg*:

```
t_T1.start(1.0);
alt {
 [] serverPort.receive(rpy_msg){ // rpy_msg received
    t_T1.stop;
    setverdict(pass);
   };
 [] serverPort.receive{ // unexpected message received
```

```
    t_T1.stop;
    setverdict(fail);
   };
  [] alt_timeout(t_T1) // timer expired
};
```

The altstep *alt_timeout* has the single alternative. If an altstep has more alternatives, they are evaluated the same way as in the statement **alt**. Once the first alternative that may proceed is selected, individual statements in this alternative are executed until the last statement in this alternative is completed, or the explicit statement **return** is encountered. The statement **return** cannot specify the return value, and it transfers control back to the statement following the altstep call within the enclosing statement **alt**.

An altstep may also have local variables, which are typically used for saving received messages and some intermediate results. Like a test case, an altstep may also use the clause **runs on** to inherit ports, timers, variables, and constants of the corresponding test component. The following altstep *alt_receive_10* uses the variable *v_count* to count the number of received *rpy_msg* messages, and also uses the clause **runs on** to inherit the port *serverPort* from the test component *ARPTester*:

```
altstep alt_receive_10(in ARPMessage rpy_msg) runs on ARPTester {
 var integer v_count := 0;
 alt{
  [] serverPort.receive(rpy_msg){ // expected message received
     v_count := v_count + 1;
     if(v_count == 10){ // expected number of replies
      setverdict(pass)
     }
     else if(v_count > 10){ // unexpected number of replies
      setverdict(fail)
     }
     else {
      repeat // repeat in order to receive the next message
     }
   };
  [] serverPort.receive{ // unexpected message received
     setverdict(fail)
   }
};
```

It is important to remember that an altstep must not change the current snapshot by the initialization of its local variables. The restrictions on operations that may be used for initializing the altstep's local variables are actually the same as the restrictions for the Boolean guards of the statement **alt**,

which we have already discussed previously. An example of the initialization that does not change the current snapshot is the statement for saving the received message into the altstep's local variable.

The altstep call has an optional block statement following it, which is executed after the altstep if any of the alternatives within the altstep are triggered. This block statement may be, for example, used to stop a running timer.

We may now use the altsteps *alt_timeout* and *alt_receive_10* to construct the statement **alt** for receiving the ten *rpy_msg* messages with the guard against unexpected messages and within the time interval bounded by the timer *t_T10*; we also use the optional statement block after the altstep *alt_ receive_10* call to stop the timer *t_T10*:

```
// create the ARP reply message
ARPMessage rpy_msg := t_ARPMessage(
 2, "00:EB:24:B2:05:C8","192.168.0.48","00:EB:24:B2:05:C0", "192.168.0.40"
);
t_T10.start(10.0);
alt {
 [] alt_receive_10(rpy_msg){ // receive 10 rpy_msg
    t_T10.stop
   };
 [] alt_timeout(t_T10) // timer expired
};
```

The reception of the ten *rpy_msg* messages is performed by the altstep *alt_receive_10*, which contains the statement **repeat**. Note that the inner of the two nested **alt** statements would be repeated. More precisely, the statement **alt** defined within the altstep *alt_receive_10* would be repeated.

We may use the operation **return** to end the execution of altstep at the desired point. The operation **return** returns the control to the enclosing statement **alt**, and then the optional block statement following the altstep call is executed. Alternatively, we may use the operation **break** to end the execution of the altstep at some point. The operation **break** returns control to the statement following the enclosing statement **alt**. Note that the optional block statement following the altstep call would not be executed in this case.

So, we should remember that the operation **return** leaves the enclosing altstep, whereas the operation **break** leaves the enclosing statement **alt** from which the altstep was called.

Normally, some more simple altsteps appear in many **alt** statements. The altstep *alt_timeout*, which we introduced earlier, is a typical example of such an altstep. Another typical example is the following altstep named *alt_ receive_any*, which is typically used to catch unexpected messages:

```
altstep alt_receive_any() runs on ARPTester {
 [] any port.receive {
```

```
 setverdict(fail);
 };
};
```

We may avoid adding such frequently used altstep to all the **alt** statements in our test suite by using them as **default altsteps**. Although they have a special name, we define the default altsteps exactly the same way we define the nondefault altsteps that we have used so far, such as the altstep *alt_receive_any* we have defined above.

The **default altstep** is an altstep that has been activated by the operation **activate**, and it remains the default altstep until it is deactivated by the operation **deactivate**. The operation **activate** adds the default altstep at the head of the list of default altsteps. This list of default altsteps is implicitly added at the end of each **alt** statement in the test suite. The operation **deactivate** removes the specified default altstep from the list of default altsteps.

Since the list of the default altsteps is evaluated from head to tail, the default altstep *Y* that has been activated after the default altstep *X* has a higher priority than the altstep *X*. In other words, if the altstep *Z* has been activated last and the altstep *A* has been activated first, *Z* would have the highest priority and *A* would have the lowest priority. In practice, we use this rule such that we activate the more general default altsteps before the more specific default altsteps, thus the latter would have a higher priority.

The parameter of the operation **activate** is the altstep together with its arguments, and the return value of the operation **activate** is the reference to the activated default altstep, which is of the type **default**. The parameter of the operation **deactivate** is the reference to the default altstep that should be deactivated.

There is one important rule related to the default altstep's call by reference parameters, i.e., **out** and **inout** parameters. Values and templates cannot be **out** or **inout** parameters of default altsteps. The reason for introducing this rule is that a value or a template passed by a reference to the default altstep might not exist at the time when the default altstep has to be executed.

Another important rule is that timers and ports may be passed as **inout** parameters to the default altsteps. Alternatively, a default altstep may inherit timers and ports of the test component that it **runs on**. In other words, in order to provide access to timers and ports within the default altstep, we may either pass them as **inout** parameters or we may provide access to the test component's timers and ports by using the clause **runs on**.

In the following two examples, we adapt the previously introduced test case *tc_resolve_guraded* by using the default altsteps *alt_timeout* and *alt_receive_any*. We do the adaptation in two steps: In the first step we just introduce the default altsteps and then in the second step we use the convention of the implicit expansion of stand-alone blocking statements into the corresponding statement **alt**, but in the reverse order, to further shorten the final test case. The result of the first step of adaptation is the following test case:

```
testcase tc_resolve_default1() runs on ARPTester {
 timer t_T1;
 var default v_ref1, v_ref2;
 // activate the default altsteps – more general first
 v_ref1 = activate(alt_timeout(t_T1));
 v_ref2 = activate(alt_receive_any());
 // create the ARP request message
 ARPMessage req_msg := t_ARPMessage(
  1,"00:EB:24:B2:05:C0","192.168.0.40",0,"192.168.0.48"
 );
 // send the ARP request message
 serverPort.send(req_msg);
 // part 2
 // create the ARP reply message
 ARPMessage rpy_msg := t_ARPMessage(
  2,"00:EB:24:B2:05:C8","192.168.0.48",
  "00:EB:24:B2:05:C0","192.168.0.40"
 );
 // start the timer t_T1 with duration 1s
 t_T1.start(1.0);
 // use the statement alt with 3 alternatives (2 are implicit)
 alt {
  [] serverPort.receive(rpy_msg){ // rpy_msg received
     t_T1.stop;
     setverdict(pass);
  };
  // alt_receive_any is implicitly considered first
  // alt_timeout is implicitly considered second
 }
 // deactivate the default altsteps
 deactivate(v_ref1);
 deactivate(v_ref2);
};
```

Remember that we intentionally activate the more specific default altsteps later than the more general, so that the former have a higher priority. In this example, we activated the default altstep *alt_receive_any* after the default altstep *alt_timeout*, so that the unexpected message may be cached before the timer *t_T1* expires.

Next, we transform the statement **alt** with a single alternative into the corresponding stand-alone blocking statement. The resulting test case is the following:

```
testcase tc_resolve_default2() runs on ARPTester {
 timer t_T1;
 var default v_ref1, v_ref2;
```

```
 // activate the default altsteps – more general first
 v_ref1 = activate(alt_timeout(t_T1));
 v_ref2 = activate(alt_receive_any());
 // create the ARP request message
 ARPMessage req_msg := t_ARPMessage(
  1, "00:EB:24:B2:05:C0","192.168.0.40",0,"192.168.0.48"
 );
 // send the ARP request message
 serverPort.send(req_msg);
 // part 2
 // create the ARP reply message
 ARPMessage rpy_msg := t_ARPMessage(
  2,"00:EB:24:B2:05:C8","192.168.0.48",
  "00:EB:24:B2:05:C0","192.168.0.40"
 );
 // start the timer t_T1 with duration 1s
 t_T1.start(1.0);
 // this stand-alone blocking statement is implicitly expanded
 // into the corresponding single-alternative alt statement
 serverPort.receive(rpy_msg){ // rpy_msg received
   t_T1.stop;
   setverdict(pass);
 };
 // deactivate the default altsteps
 deactivate(v_ref1);
 deactivate(v_ref2);
};
```

Obviously, by using the default altsteps we may get rather compact code. However, the disadvantage of using the default altsteps is that we may forget which default altsteps are currently active and their order of activation, especially if we often activate and deactivate them. The code using the default altsteps may be hard to understand and maintain, so we should use the default altsteps carefully.

Functions in TTCN-3 may be also used to specify communication behavior, and they may contain all the kinds of statements that we have introduced so far. Unlike the altsteps that must start with the statement **alt** at the topmost level, the functions may start with any statement, including, for example, the statement **send**.

In the following example, we define the function *sendReqBurst* whose parameter is the number of ARP requests to be sent (*p_noReqs*) and that runs on the test component *ARPTester*:

```
function sendReqBurst(in integer p_noReqs) runs on ARPTester {
 // create the ARP request message
```

```
 ARPMessage req_msg := t_ARPMessage(
  1, "00:EB:24:B2:05:C0", "192.168.0.40", 0, "192.168.0.48"
 );
 for(var integer i := 0; i < p_noReqs; i++) {
  serverPort.send(req_msg)
 };
 return;
};
```

The function *sendReqBurst* has access to the port *serverPort* because it **runs on** the test component *ARPTester*, which comprises this port. Alternatively, we may pass the port as an **inout** parameter of a function. The function *sendReqBurst* does not have a return value, and we use the statement **return** at the end of the function to explicitly indicate the end of the function.

Next, we adapt the previously introduced test case *tc_resolve_robustness* to use the newly introduced function *sendReqBurst*:

```
testcase tc_resolve_robustness_fun() runs on ARPTester {
 timer t_T10;
 // call the function to send the burst of 10 requests
 sendReqBurst (10);

 // part 2
 // create the ARP reply message
 ARPMessage rpy_msg := t_ARPMessage(
  2,"00:EB:24:B2:05:C8","192.168.0.48",
  "00:EB:24:B2:05:C0","192.168.0.40"
 );
 // start the timer t_T1 with duration 10s
 t_T10.start(10.0);
 // use the statement alt and repeat to receive 10 ARP replies
 alt {
  [] serverPort.receive(rpy_msg){ // rpy_msg received
     setverdict(pass);
     repeat; // repeat in order to receive the next reply
    };
  [] serverPort.receive{ // unexpected message received
     t_T10.stop;
     setverdict(fail);
    };
  [] t_T10.timeout{ // timer expired
     setverdict(fail)
    }
  }
};
```

TTCN-3 makes no distinction between the ordinary value-computing functions and the communication-behavioral functions. We may call the former functions from the latter and vice versa. Both simple and recursive function calls are allowed.

The function defined using the clause **runs on** naturally can be executed on the instance of the specified component type, but it can also be executed on the instance of the component type that is the extension of the specified component type. The extended component type must have all the timers, ports, constants, and variables of the original type and it may have additional timers, ports, constants, and variables. For example, the function *sendReqBurst* can be also executed on the test component type *ARPTesterS*, which is the extension of the component type *ARPTester*.

Analogously, the altstep defined using the clause **runs on** can be executed both on the instance of the specified component type and on the instance of the component type that is the extension of the specified component type. For example, the altstep *alt_timeout* can be executed both on the component types *ARPTester* and *ARPTesterS*.

The next restriction applies to both functions and altsteps. A function or an altstep that is defined without the clause **runs on**, cannot be called with the clause **runs on**.

Finally, we summarize similarities and differences between the functions and the altsteps. The similarities between the functions and the altsteps are as follows:

- Both may define communication behavior.
- Both may have parameters.
- Both may be defined using the clause **runs on**.
- Both may call functions and altsteps.

The differences between the functions and the altsteps are as follows:

- Altsteps can be used at the top level of the statement alt, whereas functions can only be used in the statements within alternatives or the Boolean guards.
- Altsteps without values and template parameters passed by a reference can be activated as the default altsteps, whereas functions cannot be used to specify the default behavior.
- Altsteps must start with the statement **alt**, whereas functions may start with any statement.
- Altsteps cannot use initializations of local variables that change the current snapshot, whereas functions can use local variables without any restrictions.
- Altsteps cannot have return value, whereas functions can have return value.

3.10 Examples

This section contains some examples that are related to the communication protocol design. These should help the reader to consolidate their understanding of the concepts and techniques introduced so far.

3.10.1 Example 1

This example demonstrates the procedures for connection establishment and release that are performed by two communicating processes, namely *TE1* and *TE2*. The processes *TE1* and *TE2* are specified by their statechart diagrams shown in Figures 3.47 and 3.48, respectively. The semantically equivalent SDL diagrams are shown in Figures 3.49 and 3.50, respectively.

The process *TE1* has four stable states, labeled *TE1_IDLE*, *TE1_CONNECTING*, *TE1_CONNECTED*, and *TE1_DISCONNECTING*. While the process *TE1* is in the state *TE1_IDLE*, it can receive only the message *CONNECT_req* from the user and after receiving that message, the process *TE1* sends the message *CONNECT_ind* to the process *TE2*, and evolves to its next stable state *TE1_CONNECTING*. In that state, the process may receive one of two possible input messages, namely *CONNECT_conf* or *CONNECT_reject*. In the former case, the process moves to the stable state *TE1_CONNECTED*, whereas in the latter case, it evolves to its initial stable state *TE1_IDLE*.

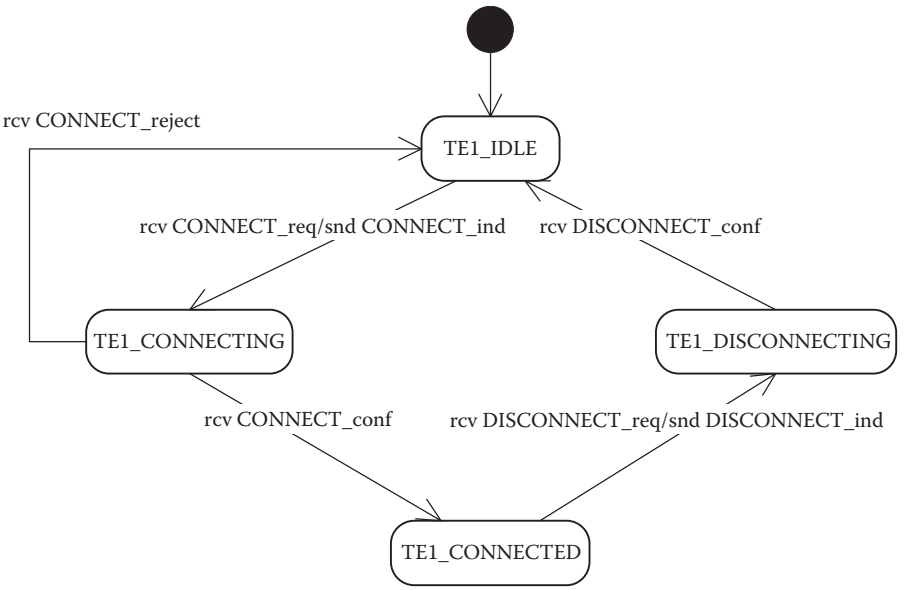


FIGURE 3.47
Statechart diagram of the process *TE1*.

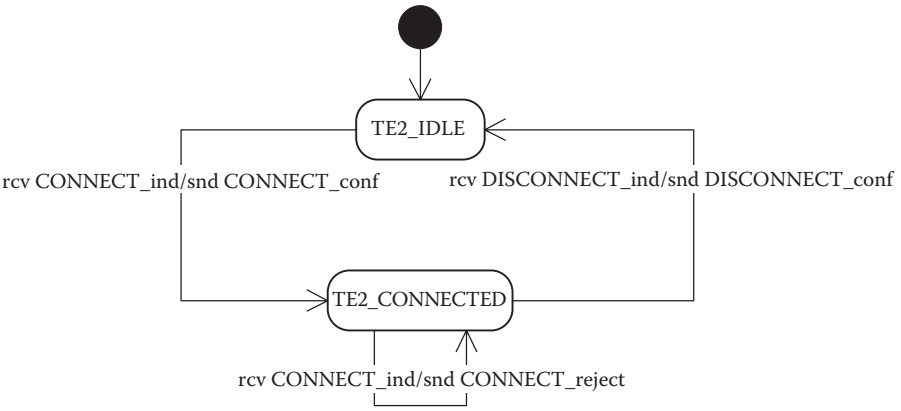


FIGURE 3.48
Statechart diagram of the process *TE2*.

In its stable state *TE1_CONNECTED*, the process *TE1* may receive the message *DISCONNECT_req* from the user. In that case, it sends the message *DISCONNECT_ind* to the process *TE2* and evolves to the stable state *TE1_DISCONNECTING*. From that stable state, it returns to its initial stable state *TE1_IDLE* after receiving the message *DISCONNECT_conf* from its peer process *TE2*.

The SDL diagram specification of the process *TE2* is much simpler because it comprises only two stable states, namely, *TE2_IDLE* and *TE2_CONNECTED*. In the former state, the process *TE2* may receive only the message *CONNECT_ind*, to which it replies by the message *CONNECT_conf* and after that, it evolves to the state *TE2_CONNECTED*. In the latter state, the process may receive one of two possible messages, *CONNECT_ind* or *DISCONNECT_ind*. In the former case, the process *TE2* replies with the message *CONNECT_reject* and remains in its current state. In the latter case, it replies with the message *DISCONNECT_conf* and goes back to its initial state *TE2_IDLE*.

The scenario of a successful connection establishment and release is illustrated by the MSC chart shown in Figure 3.51. The top of the chart shows the communicating entities, the human user, and the program processes *TE1* and *TE2*. The vertical lines are drawn from the rectangular graphical symbols down to the bottom of the sheet. The time advances in the same direction.

The connection establishment procedure starts when the user sends the message *CONNECT_req* to the process *TE1* (this event is noted by the arrow drawn from the vertical line labeled *USER* to the vertical line labeled *TE1*), which in turn sends the message *CONNECT_ind* to the process *TE2*. The process *TE2*, in turn, replies with the message *CONNECT_conf*. Upon receipt of the message *CONNECT_conf*, the process *TE1* forwards it to the user. This completes the connection establishment procedure. The next communication phase is normally used for the desired data transfer. Because of that, it is most frequently referred to as a data transfer phase.

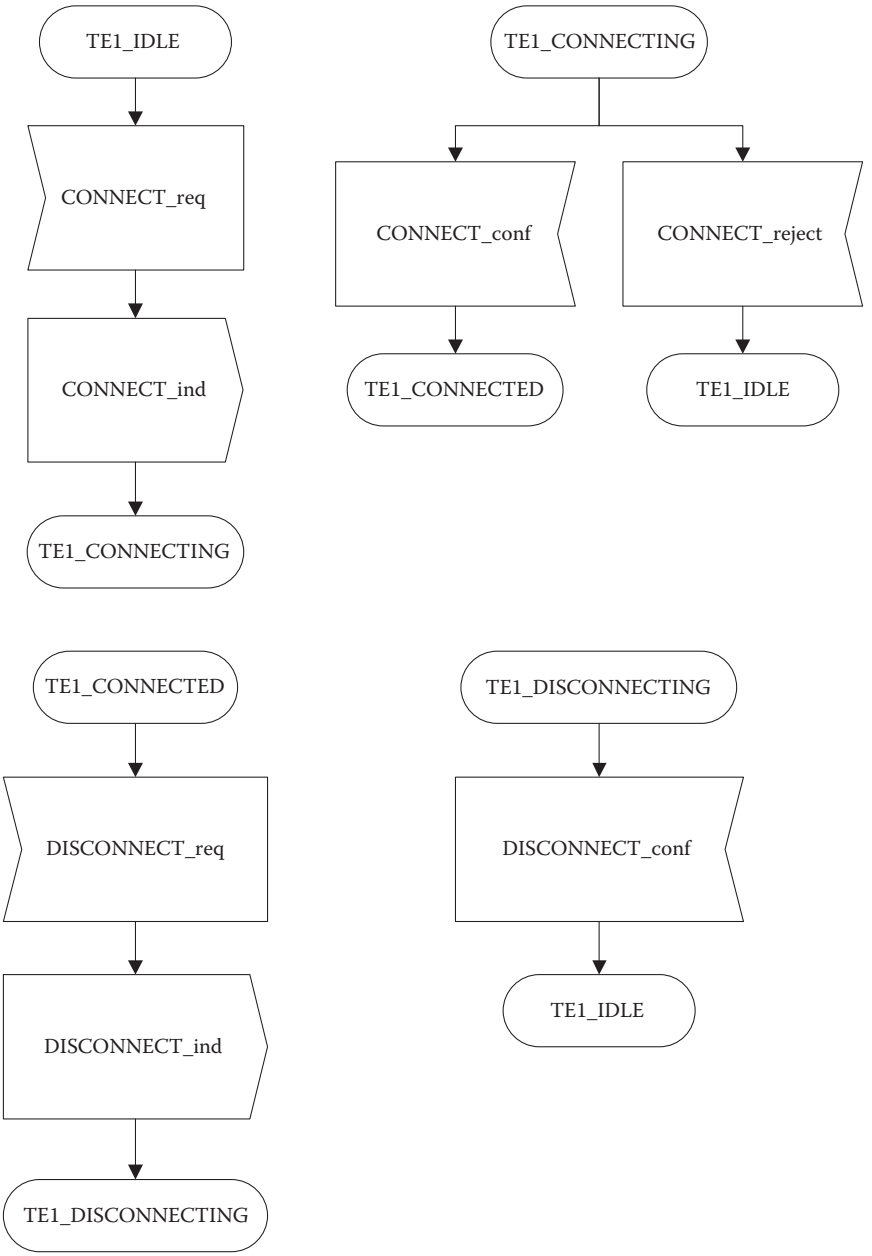


FIGURE 3.49
SDL diagram of the process *TE1*.

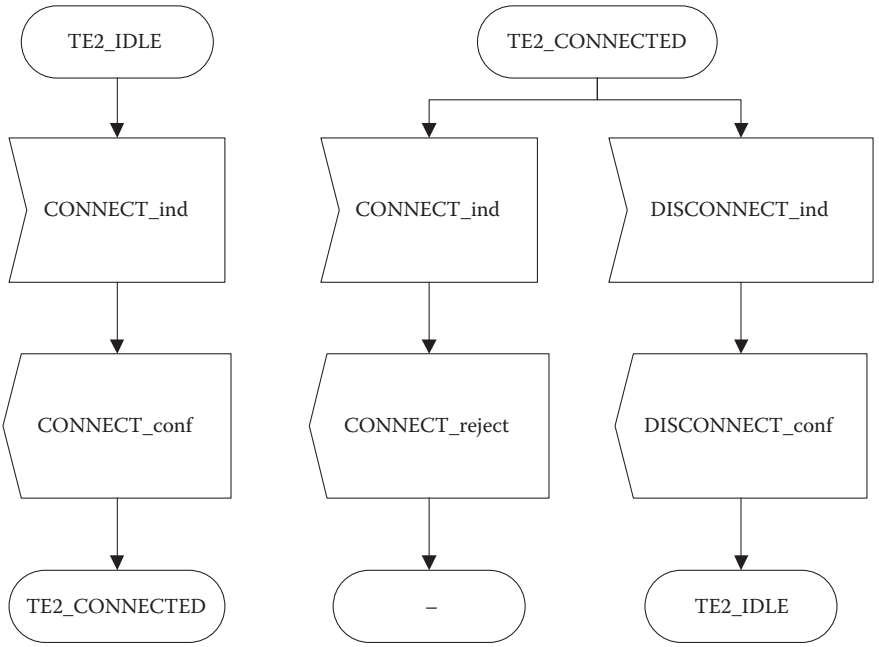


FIGURE 3.50
SDL diagram of the process *TE2*.

The connection release procedure starts when the user sends the message *DISCONNECT_req* to the process *TE1*, which translates it to the message *DISCONNECT_ind* and sends it to the process *TE2*, which, in turn, replies by the message *DISCONNECT_conf*. Upon receipt of the message *DISCONNECT_conf*, the process *TE1* forwards it to the user. This completes the connection release procedure.

Next, we develop a simple TTCN-3 test suite specification for this example, which comprises two test cases. We start by defining the new types *Address*, *Data*, and *Msg*:

```
type enumerated Code (
 CONNECT_req, CONNECT_ind,
 CONNECT_conf, CONNECT_reject,
 DISCONNECT_req, DISCONNECT_ind,
 DISCONNECT_conf, DISCONNECT_reject
);
type integer Adress;
type integer Data;
type record Msg {
 Code code;
 Address source_address;
```

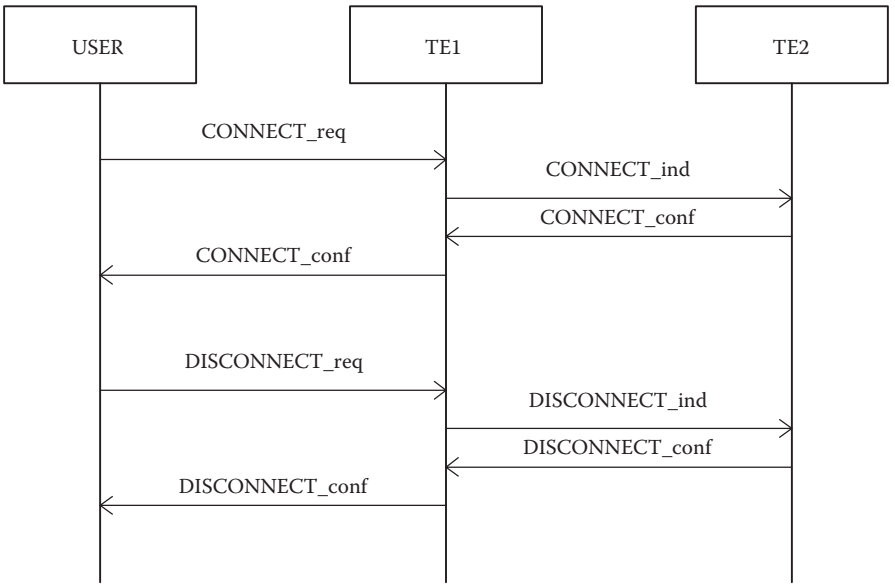


FIGURE 3.51
Successful connection establishment and release MSC.

```
 Address destination_address;
 Data user_data;
}
```

Then by using the message type *Msg*, we define the suitable parametrized templates *t_request* and *t_indication*:

```
template Msg t_request(Code p_code, Address p_src, Address p_dst) := {
 code := p_code,
 source_address := p_src,
 destination_address:= p_dst,
 user_data := ?
}

template Msg t_indication(Code p_code, Address p_src, Address p_dst) := {
 code := p_code,
 source_address := p_src,
 destination_address:= p_dst,
 user_data := ?
}
```

Let's assume that *USER*, *TE1*, and *TE2*, are assigned the addresses 0, 1, and 2, respectively. Let's also assume that the test system plays the role of *TE1*, and that it communicates with *USER* and *TE2* over the ports *pt_user* and *pt_te2*, respectively. Finally, we assume that both of these ports are of type *PortTS*, which are defined as follows:

```
type port PortTS {
 inout Msg
}
```

Our simple test suites use a single test component named *ComponentTS* to execute test cases, and *ComponentTS*, in turn, uses two previously mentioned communication ports to communicate with *USER* and *TE2*:

```
type component ComponentTS {
 port PortTS pt_user;
 port PortTS pt_te2
}
```

The first test case tests the connection establishment phase of the communication, which correspond to the top half of the MSC chart shown in Figure 3.51:

```
testcase tc_no1() runs on ComponentTS {
 pt_user.receive( t_request(CONNECT_req, 0, 1) );
 pt_te2.send( t_indication(CONNECT_ind, 1, 2) );
 alt {
 [] pt_te2.receive( t_indication(CONNECT_conf, 2, 1) ) {
 pt_user.send( t_indication(CONNECT_conf, 1, 0) );
 setverdict( pass );
 }
 [] pt_te2.receive( t_indication(CONNECT_reject, 2, 1) ) {
 setverdict( inconc );
 }
 }
 stop;
}
```

The second test case tests both the connection establishment phase and the connection release phase of the communication, which correspond to the complete MSC chart shown in Figure 3.51:

```
testcase tc_no2() runs on ComponentTS {
 // check the connection establishment phase
 pt_user.receive( t_request(CONNECT_req,0,1) );
 pt_te2.send( t_indication(CONNECT_ind,1,2) );
 alt {
  [] pt_te2.receive( t_indication(CONNECT_conf,2,1) ) {
     pt_user.send( t_indication(CONNECT_conf,1,0) );
     // the connection is successfully established
    }
```

```
 [] pt_te2.receive( t_indication(CONNECT_reject,2,1)) {
    setverdict(inconc);
    stop
   }
}
 // check the connection release phase
 pt_user.receive( t_request(DISCONNECT_req, 0, 1) );
 pt_te2.send( t_indication(DISCONNECT_ind, 1, 2) );
 alt {
  [] pt_te2.receive( t_indication(DISCONNECT_conf, 2, 1) ) {
     pt_user.send( t_indication(DISCONNECT_conf, 1, 0) );
     // the connection is successfully released
     setverdict( pass );
    }
  [] pt_te2.receive {
     // receive any other message
     setverdict( fail );
    }
 }
 stop;
}
```

We may execute both of these test cases by using the following control part:

```
control {
 execute( tc_no1() )
 execute( tc_no2() )
}
```

The reader is encouraged to play more with this simple example. For example, we can change the previous example so that before the existing connection is established, the process *User* checks if the process *TE1* is ready for the communication. The MSC chart that specifies a new connection establishment procedure is shown in Figure 3.52.

3.10.2 Example 2

Figure 3.53 shows a hypothetical computer network with a star topology. Three terminal nodes (N1, N2, and N3) are connected to one transit node (TN). The routing table residing in TN is shown in Figure 3.53 to the right of TN. Terminal nodes generate messages for other terminal nodes in the network. Depending on the value of the message parameter (1, 2, or 3), a transit node delivers the message to its destination by sending it to the corresponding port (A, B, or C).

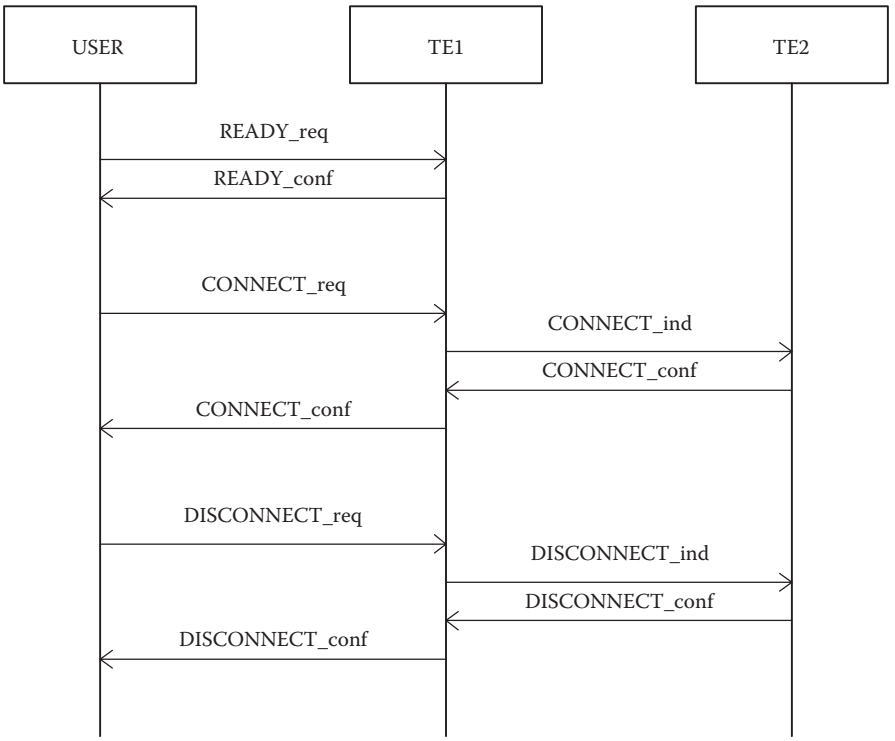


FIGURE 3.52
New connection establishment procedure MSC.

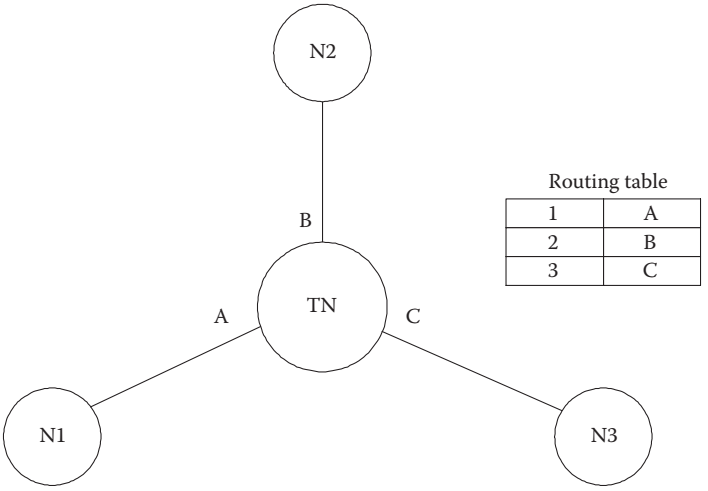


FIGURE 3.53
Hypothetical star network with one transit and three terminal nodes.

The communication process that resides in the terminal node of the network is specified by the statechart diagram shown in Figure 3.54. The process that executes in the transit node is described by the statechart diagram shown in Figure 3.55. The semantically equivalent SDL diagrams are shown in Figures 3.56 and 3.57, respectively.

The process that runs in the terminal node of the network has two stable states, *N123_IDLE* and *N123_MSG_SENT*. The state transition is initiated by the user message *MSG_req*. The process returns to its initial state after the reception of one of three possible messages, namely, *MSG_conf*, *MSG*, or *MSG_reject*. The process that resides in the transit node of the network has a single state, *TN_IDLE*. This process routes the input message toward its destination.

Figure 3.58 shows the scenario of a successful message delivery. The node N1 sends the correct message to the node N3 over the node TN. The user is informed about the successful delivery by the message *MSG_conf*. Figure 3.59 shows the scenario of an unsuccessful message delivery. The node N1 has

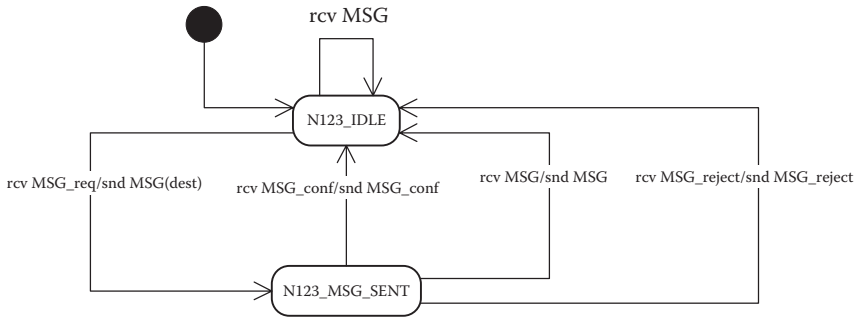


FIGURE 3.54
Statechart diagram of the process that runs in a terminal node of the network.

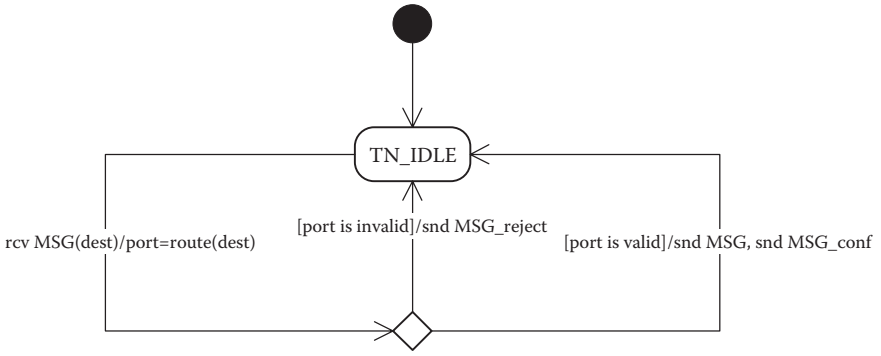


FIGURE 3.55
Statechart diagram of the process that resides in the transit node of the network.

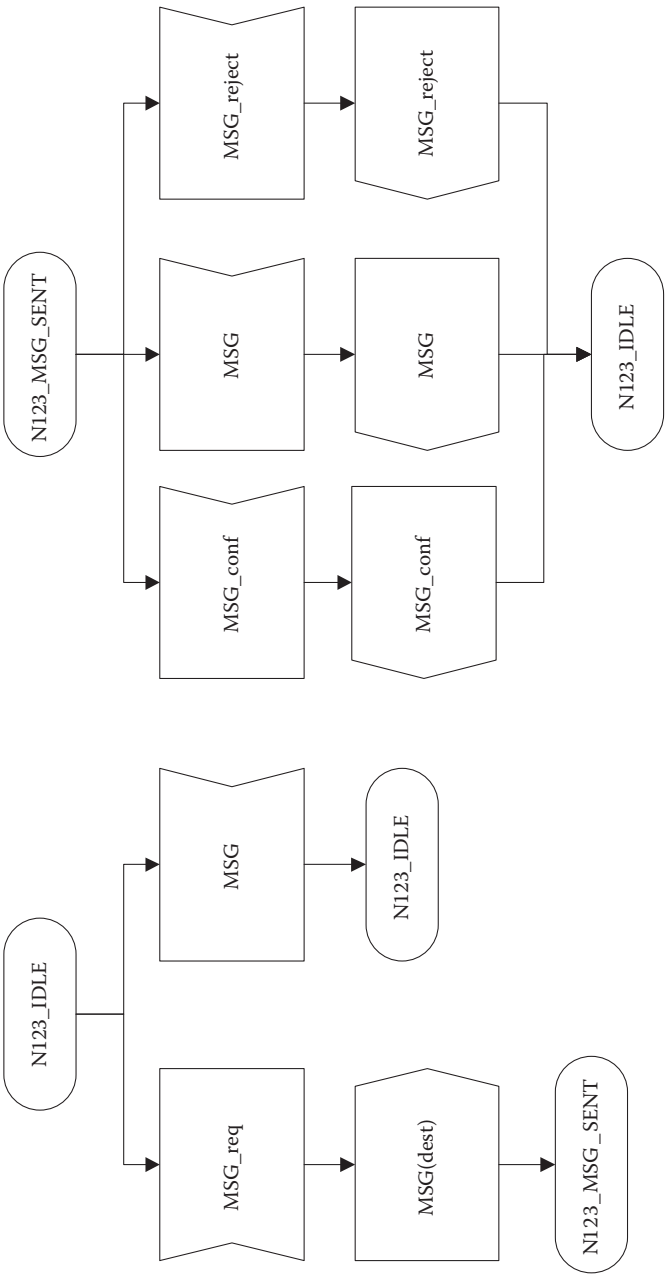


FIGURE 3.56
SDL diagram of the process that runs in a terminal node of the network.

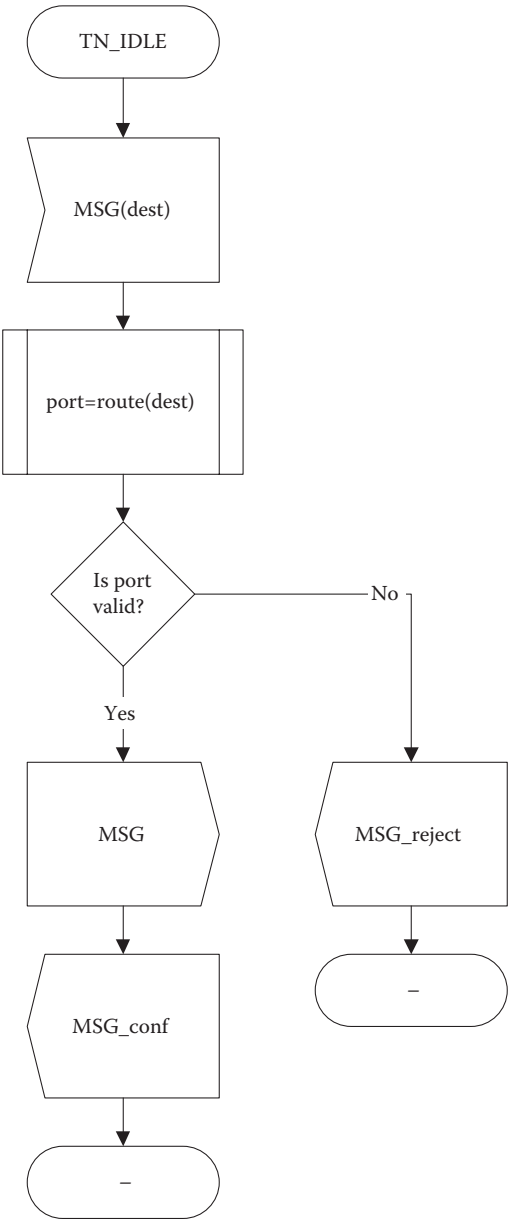


FIGURE 3.57
SDL diagram of the process that resides in the transit node of the network.

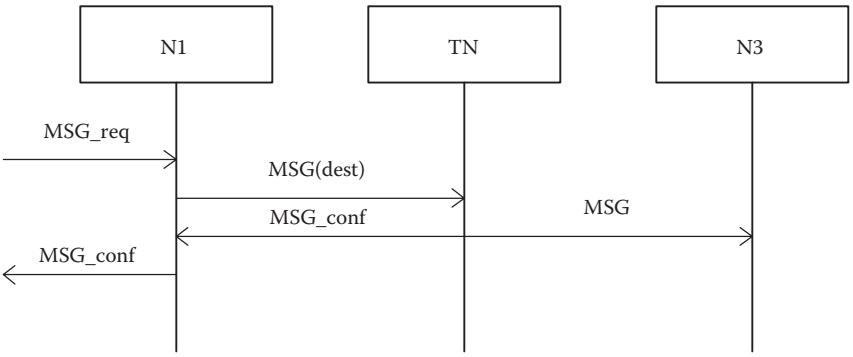


FIGURE 3.58
Successful message delivery MSC.

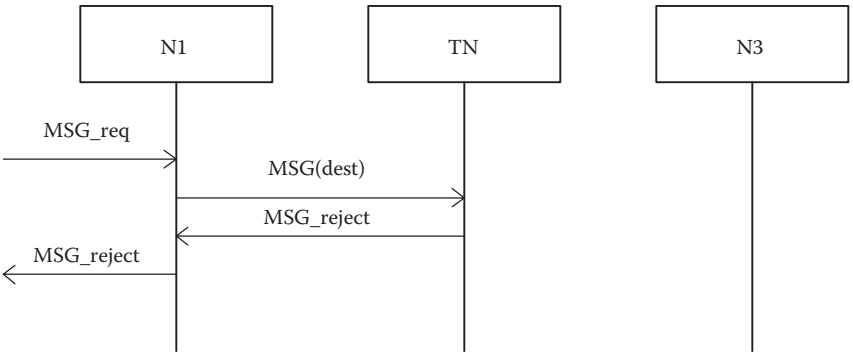


FIGURE 3.59
Unsuccessful message delivery MSC.

sent the message to the unknown destination, which has been rejected from the node TN by the message *MSG_reject*.

Next, we develop a simple TTCN-3 test suite specification for this example, which comprises two test cases. We start by defining the new types *Address*, *Data*, and *Msg*:

```
type enumerated Code (
 MSG_req,
 MSG_conf,
 MSG_reject
);
type integer Adress;
type integer Data;
type record Msg {
 Code code;
```

```
 Address destination_address;
 Data user_data;
}
```

Then by using the message type *Msg* we define the suitable parametrized templates *t_request* and *t_response*:

```
template Msg t_request(Code p_code, Address p_dst) := {
 code := p_code,
 destination_address:= p_dst,
 user_data := ?
}

template Msg t_response(Code p_code, Address p_dst) := {
 code := p_code,
 destination_address:= p_dst,
 user_data := ?
}
```

Let's assume that the test system plays the role of N1 in Figures 3.58 and 3.59, and that it communicates with USER and TN over the ports *pt_user* and *pt_tn*, respectively. We assume that both of these ports are of type *PortN*, which is defined as following:

```
type port PortN {
 inout Msg
}
```

Our simple test suites use a single test component named *ComponentN* to execute test cases, and *ComponentN*, in turn, uses two previously mentioned communication ports to communicate with USER and TN:

```
type component ComponentTS {
 port PortN pt_user;
 port PortN pt_tn
}
```

The first test case tests the successful delivery of the correct message from N1 to N3, in accordance with the MSC chart shown in Figure 3.58:

```
testcase tc_no1() runs on ComponentN {
 pt_user.receive( t_request(MSG_req, 3) );
 pt_n1.send( t_request(MSG_req, 3) );
 alt {
  [] pt_n1.receive( t_response(MSG_conf, 3) ) {
     pt_user.send( t_response(MSG_conf, 3) );
```

```
    setverdict( pass );
   }
 [] pt_n1.receive( t_response(MSG_reject, 3) ) {
    pt_user.send( t_response(MSG_reject, 3) );
    setverdict( fail );
   }
}
stop;
}
```

The second test case tests the successful drop of the incorrect message from N1 to non-existing N4, in accordance with the MSC chart shown in Figure 3.59:

```
testcase tc_no2() runs on ComponentN {
pt_user.receive( t_request(MSG_req, 4));
pt_n1.send( t_request(MSG_req, 4));
alt {
 [] pt_n1.receive( t_response(MSG_conf, 4)) {
    pt_user.send( t_response(MSG_conf, 4));
    setverdict( fail );
   }
 [] pt_n1.receive( t_response(MSG_reject, 4) ) {
    pt_user.send( t_response(MSG_reject, 4) );
    setverdict( pass );
   }
}
stop;
}
```

We may execute both of these test cases by using the following control part:

```
control {
execute( tc_no1() )
execute( tc_no2() )
}
```

The reader is encouraged to play more with this example. One interesting direction of generalization would be to consider a more complex network, such as the one shown in Figure 3.60.

3.10.3 Example 3

This example illustrates reliable packet delivery based on message acknowledgment. Each communication process expects the acknowledgment of the

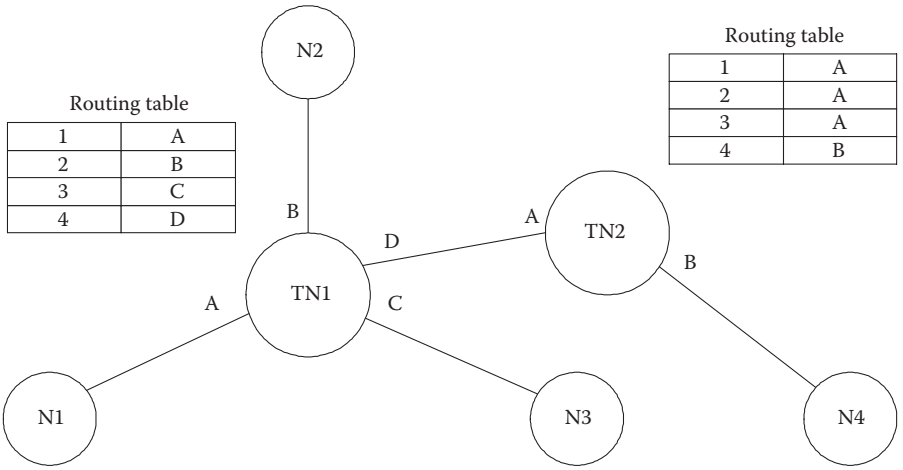


FIGURE 3.60
Topology of a more complex hypothetical network.

message that it has previously sent. If the acknowledgment is not received within the limited period of time, the corresponding timer will expire, the process will assume that the message or its acknowledgment have been lost, and the process will retransmit the message once again.

The statechart diagram and the SDL diagram of the process are shown in Figures 3.61 and 3.62, respectively. The process has two stable states, *FSM_IDLE* and *FSM_MSG_SENT*. In its initial state, the process starts the timer *T1*, sends the message with the sequence number *SN*, and evolves into its next stable state *FSM_MSG_SENT*. In that state, the process either receives the acknowledgment, stops the timer *T1*, and returns to its initial state, or the timer *T1* expires and, in turn, the process retransmits the message.

In any state (*FSM_IDLE* or *FSM_MSG_SENT*), the process can receive a message from its peer process. The process acknowledges the message if the sequence number of the message is valid (in communication protocols, the process would normally maintain the counter of the next expected message in a sequence by incrementing its contents for each received message—a validity check in this context would be to compare the sequence number in the received message with the contents of this counter). If the sequence number, *RN*, of the message is invalid, the process throws the message away.

Figure 3.63 illustrates two scenarios of the communication between two peer processes. The MSC on the left in Figure 3.63 shows a successful message delivery. The process *FSM1* sends the message *M1* to the process *FSM2*, which in turn sends the acknowledgment *ACK* to the process *FSM1*.

The MSC on the right in Figure 3.63 shows a more complex scenario of successful message retransmission after the unsuccessful first message delivery attempt. The process *FSM1* sends the message *M1*, the process *FSM2* receives it and sends its acknowledgment *ACK*, but gets lost. The timer *T1*

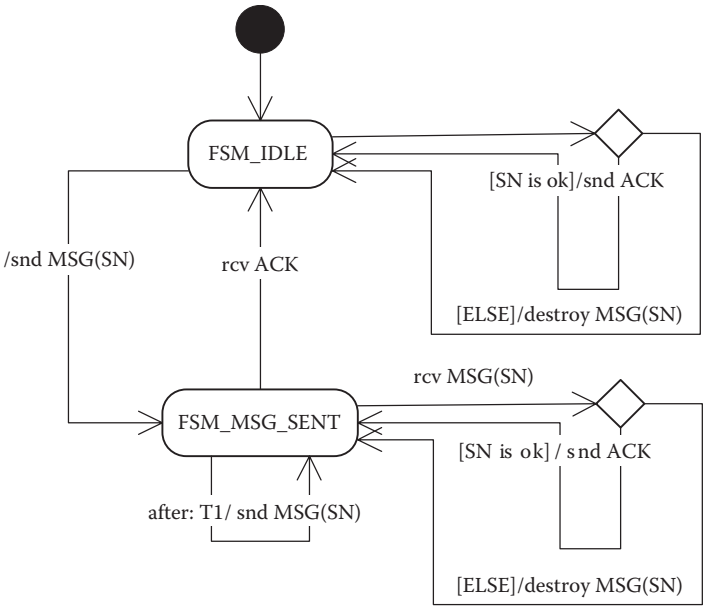


FIGURE 3.61
Statechart diagram of the communicating process that provides the reliable message delivery based on the retransmission scheme.

expires and the process *FSM1* retransmits the message *M1*. The process *FSM2* receives it and sends its acknowledgment *ACK*, which is successfully received by *FSM1*.

3.10.4 Example 4

This example illustrates the sliding window concept, which provides a reliable and efficient transport service. Voluminous literature can be found that addresses this topic (Halsall, 1988). The design shown here is based on the *Go-back-N* retransmission mechanism. It also supports the robust frame acknowledgment procedure (one ACK may acknowledge more than one frame).

The collaboration diagram in Figure 3.64 shows two distributed applications that communicate with the help of two communication objects, which are deployed at the local and remote side. The application *a1* sends the data packed into messages (*M*) to the object *p* (primary), which, in turn, encapsulates the messages into *I* (information) frames, together with its sequence number V(*s*), and sends them to the object *s* (secondary). The object *s* checks the frame *I* sequence number against the number it expects V(*r*), and if they match, it accepts the frame *I* and acknowledges it by sending the message *ACK* to the object *p*. If these numbers do not match, the object *s* rejects the received *I* frame and sends the corresponding message *NAK*. We assume

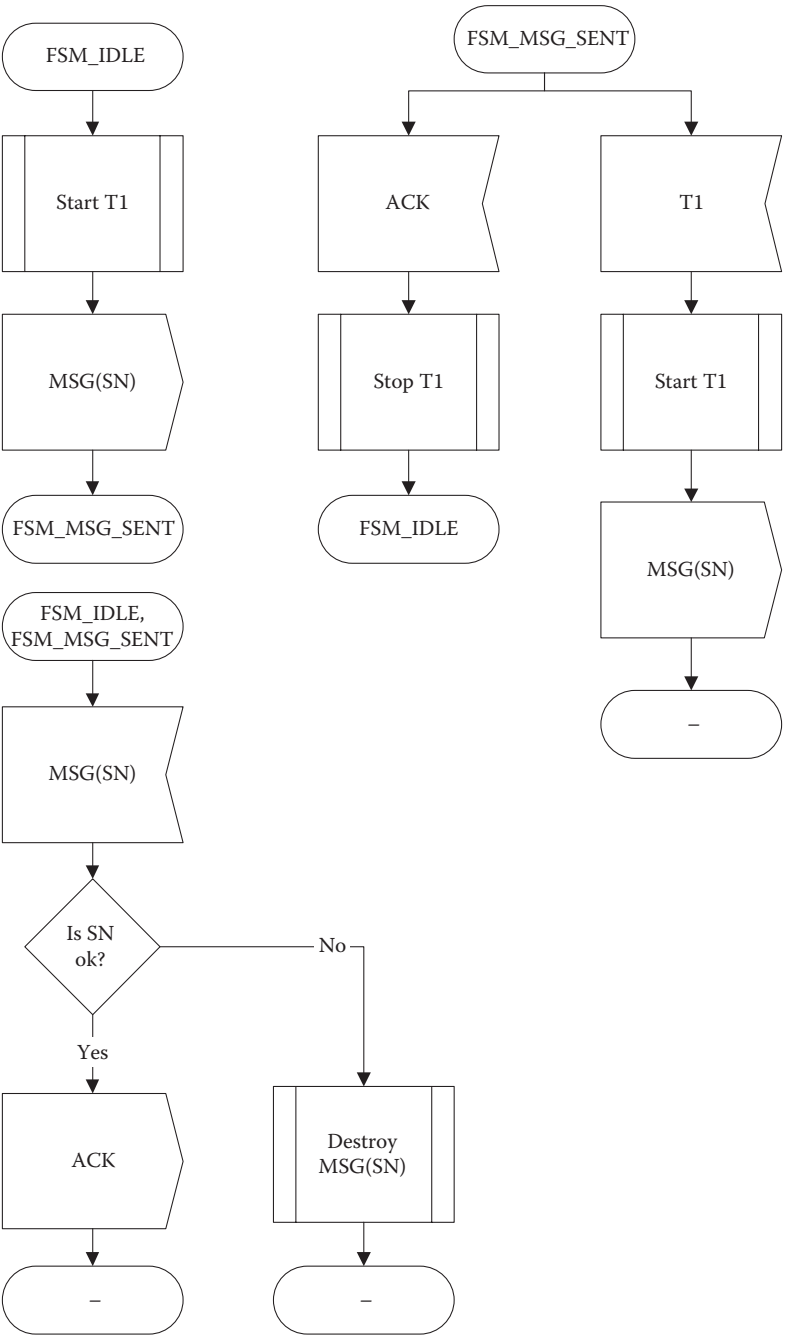


FIGURE 3.62
SDL diagram of the communicating process that provides the reliable message delivery, based on the retransmission scheme.

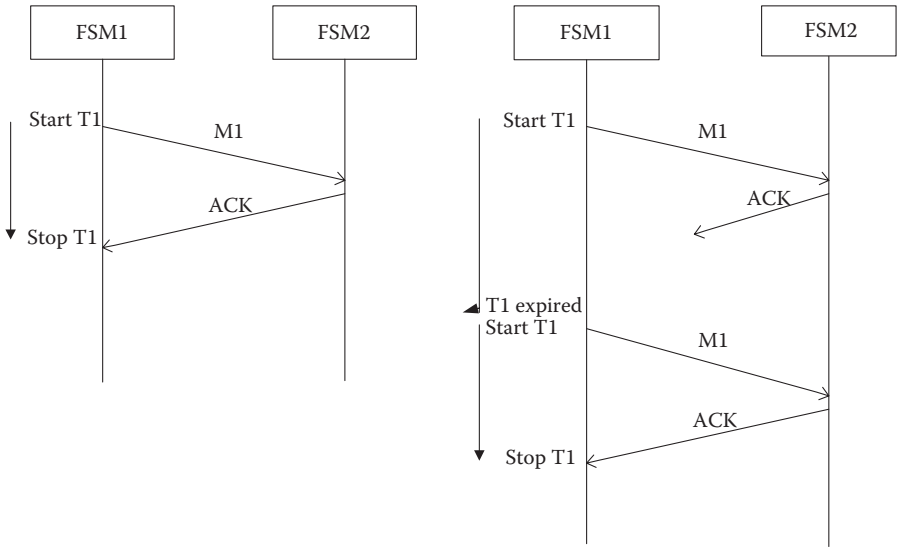


FIGURE 3.63
Example with two scenarios (with and without message retransmission).

that the numbers V(*s*) and V(*r*) are maintained in the variables *vs* and *vr*, respectively. The object *s* delivers all the correctly received messages to the remote application *a2*.

In this example, we are mainly interested in the communication protocol between the primary and the secondary side of the communication link, which is established by the corresponding communication processes, *p* and *s*. The process *p* is modeled with the activity diagram shown in Figures 3.65 and 3.66, whereas the process *s* is modeled with the activity diagram shown in Figure 3.67.

Assume that the variable *rc* holds the number of the *I* frames that were sent by the process *p* but are still not acknowledged by the process *s*. The activity diagram in Figure 3.65 starts with the transition from the initial state to the state *IDLE*. During this transition, the variables *vs* and *rc* are reset. After receiving a message *M* from the application *a1*, *p* checks if the send window is full. If the send window is not full, *p* calls the procedure *send(M)* to encapsulate *M* into *I* and sends it toward *s*. If the send window is full, *p* adds *M* to the input queue (*inputQueue*). In both cases, it returns to the state *IDLE*.

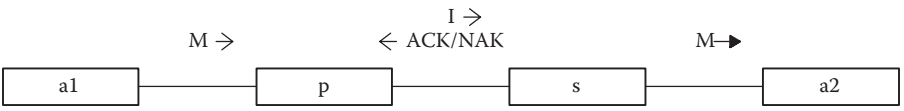


FIGURE 3.64
Example 4 collaboration diagram.

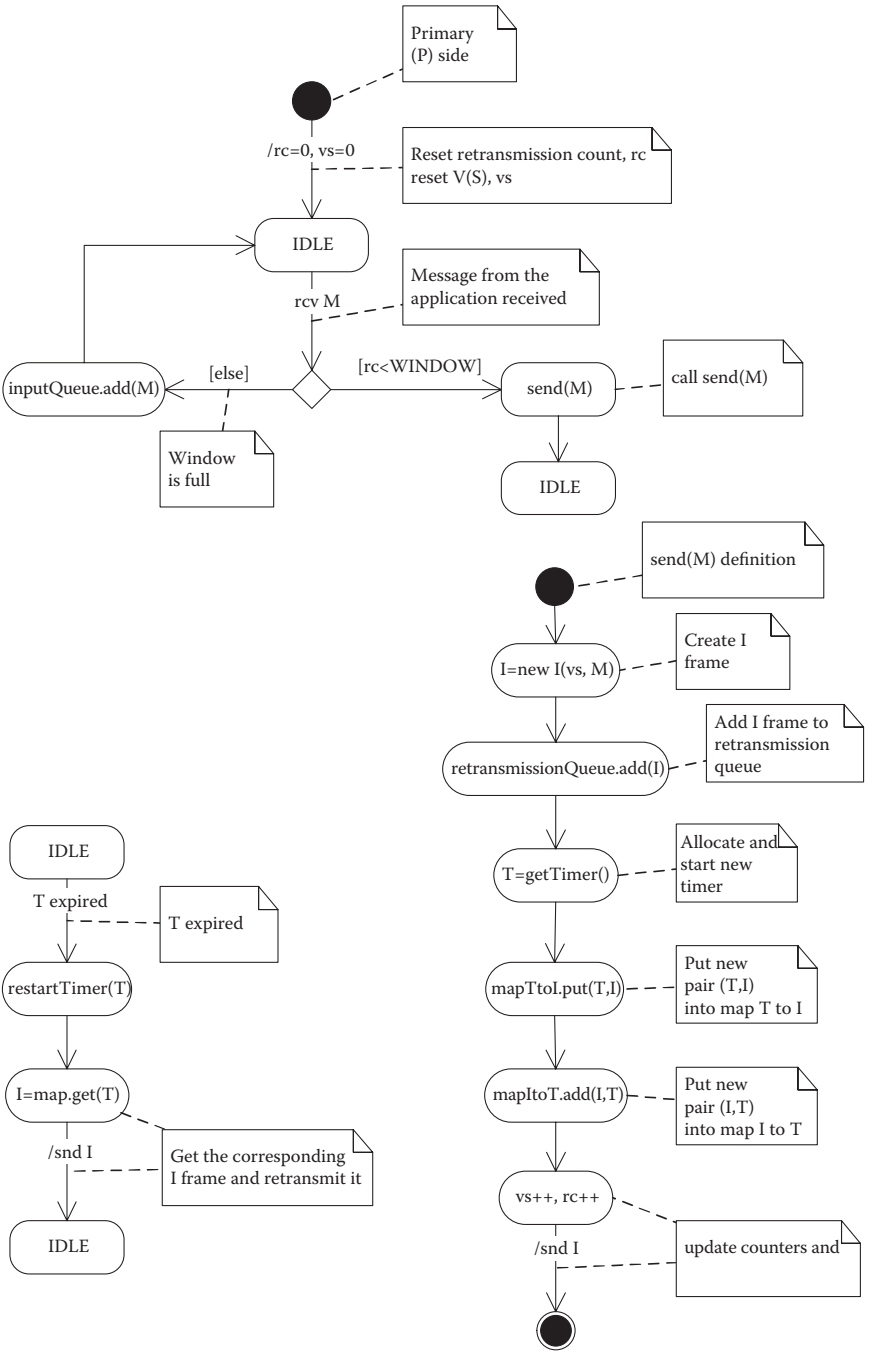


FIGURE 3.65
Example 4 activity diagram, part I.

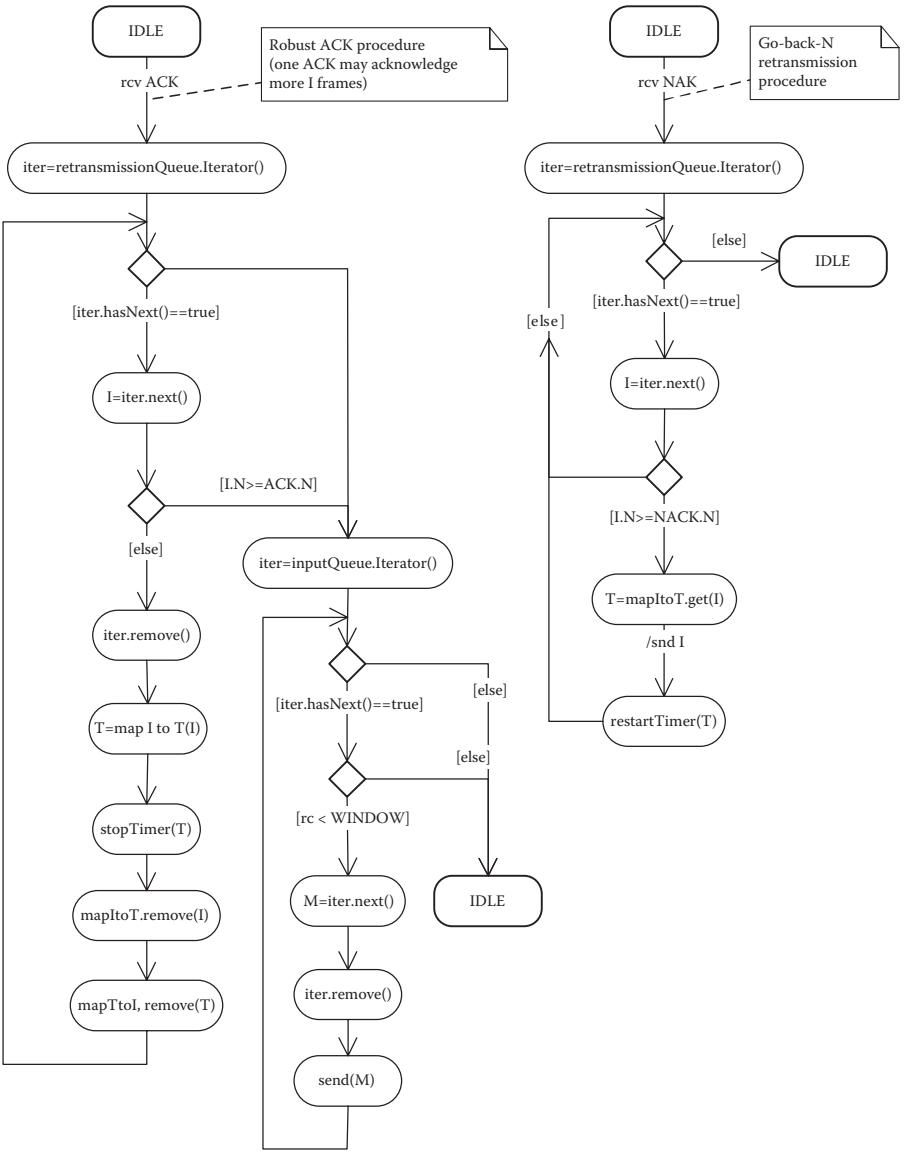


FIGURE 3.66
Example 4 activity diagram, part II.

The procedure *send(M)* first creates the frame *I* and encapsulates the current value of the variable *vs* and the message *M* in it by supplying them as arguments of the corresponding constructor. It then adds the frame to the retransmission queue (*retransmissionQueue*), allocates and starts a new timer (*T*), adds the pair (*T,I*) to the map *mapTtoI*, adds the pair (*I,T*) to the map *mapItoT*,

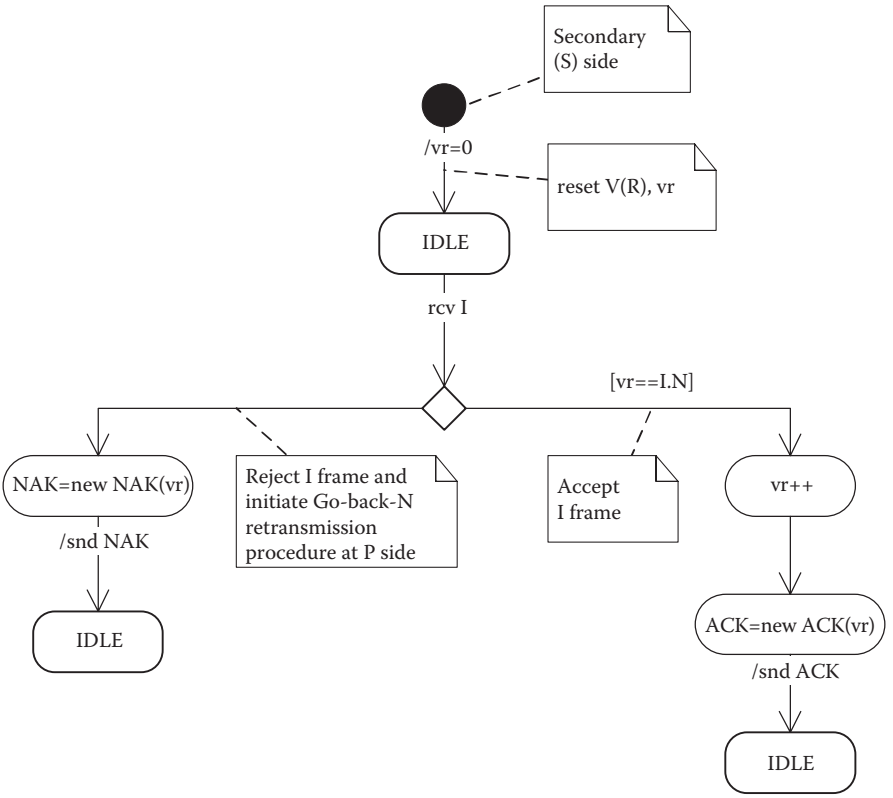


FIGURE 3.67
Example 4 activity diagram, part III.

increments *vs* and *rc*, and sends the frame *I* toward *s*. The map *mapTtoI* is used to search for the frame *I* that corresponds to the given timer *T*, whereas the map *mapItoT* is used to search for the timer *T* that corresponds to the given frame *I*. Notice that the procedure *send(M)* assigns a timer to each frame it sends. When the timer expires, *p* restarts the timer (*restartTimer(T)*), finds the corresponding frame by using the map *mapTtoI*, and retransmits the frame toward *s*.

When *p* receives the message *ACK* from *s*, it provides the iterator on the list *retransmissionQueue* and starts iterating through this list. For all the frames whose sequence number is smaller than the sequence number in the received *ACK* message, *p* finds the corresponding timer (by using the map *mapItoT*), stops it, and removes both the pair (*T*,*I*) from the map *mapTtoI* and the pair (*I*,*T*) from the map *mapItoT*.

Because some of the slots (or at least one of them) should be free after the previous iteration, *p* provides the iterator on the list *inputQueue* and starts iterating through it. It iterates while empty slots exist in the send window,

and, while iterating, it removes the messages from the input queue and sends them by calling the procedure *send(M)*, as explained previously.

If the process *p* receives the message *NAK*, it performs the *Go-back-N* retransmission procedure. Essentially, *p* scans the whole retransmission queue. For each frame whose sequence number is greater than or equal to the sequence number in the receive message *ACK*, *p* finds the corresponding timer, restarts it, and retransmits the frame toward *s*.

The activity diagram shown in Figure 3.67 models the process *s*. It starts with the triggerless transition from the initial state to the state *IDLE*. During this transition, the variable *vr* is reset. After receiving the frame *I*, *s* checks its sequence number equal to the value of the variable *vr*. If the values are the same, *s* accepts the frame by incrementing *vs*, creating the message *ACK*, and sending it to *p*. If the values are different, *s* rejects the frame by sending the message *NAK* to *p*.

Figures 3.68 through 3.70 show three typical scenarios. The sequence diagram shown in Figure 3.68 illustrates a successful frame delivery scenario. The frames *I(0)* and *I(1)* are sent through the window and are acknowledged with *ACK(1)* and *ACK(2)*, respectively. After some delay, *I(2)* is sent and it is also successfully acknowledged with *ACK(3)*.

The sequence diagram shown in Figure 3.69 illustrates the *Go-back-N* procedure. The process *p* starts by sending the frames *I(0)* and *I(1)*. The frame arrives at *s* side regularly but *I(1)* gets lost. This causes the mismatch of

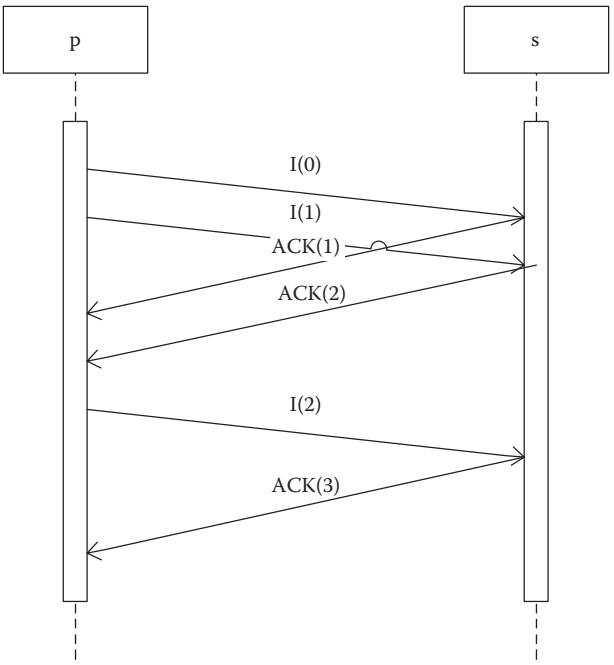


FIGURE 3.68
Example 4 MSC diagram: Successful frame delivery.

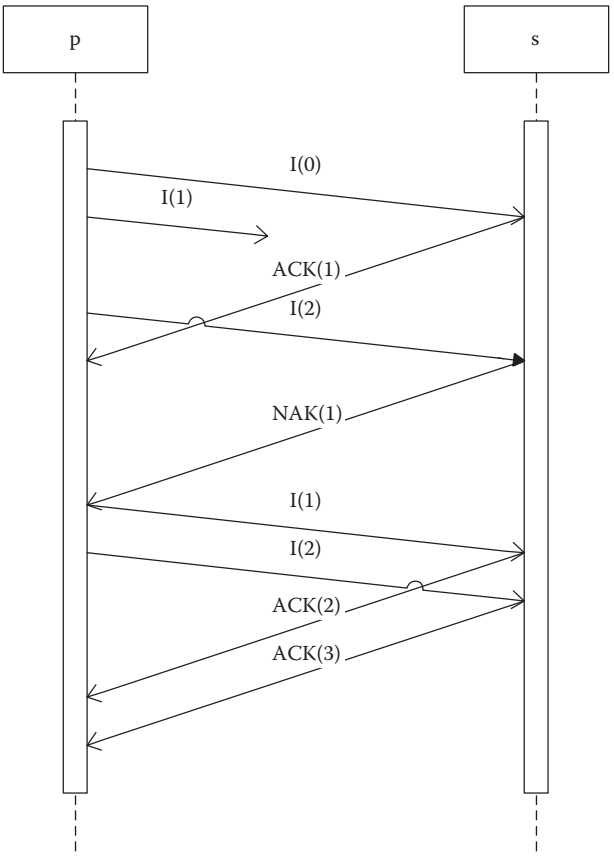


FIGURE 3.69
Example 4 MSC diagram: *Go-back-N* retransmission.

sequence numbers at the secondary side when it successfully receives *I(2)*, because the value of the variable *vr* is 1 (which indicates that *s* is awaiting *I(1)* instead of *I(2)*). Because the sequence number of the frame and the value of the variable are not the same, *s* rejects the frame by sending the message *NAK(1)*. The process *p*, in turn, retransmits both *I(1)* and *I(2)*.

The sequence diagram shown in Figure 3.70 illustrates the frame retransmission triggered by the retransmission timer. The process *p* starts again by sending *I(0)* and *I(1)* in succession. The process *s* in its turn acknowledges them by *ACK(1)* and *ACK(2)*, respectively. The message *ACK(1)* arrives successfully at the primary side, but the message *ACK(2)* gets lost. This causes the corresponding timer to expire after a while. Triggered by that event, *p* restarts the timer and retransmits the frame *I(1)*. During the second time, both *I(1)* and the corresponding *ACK(2)* are successfully transferred over the communication link. After receiving *ACK(2)*, *p* stops the timer and removes *I(1)* from the retransmission queue.

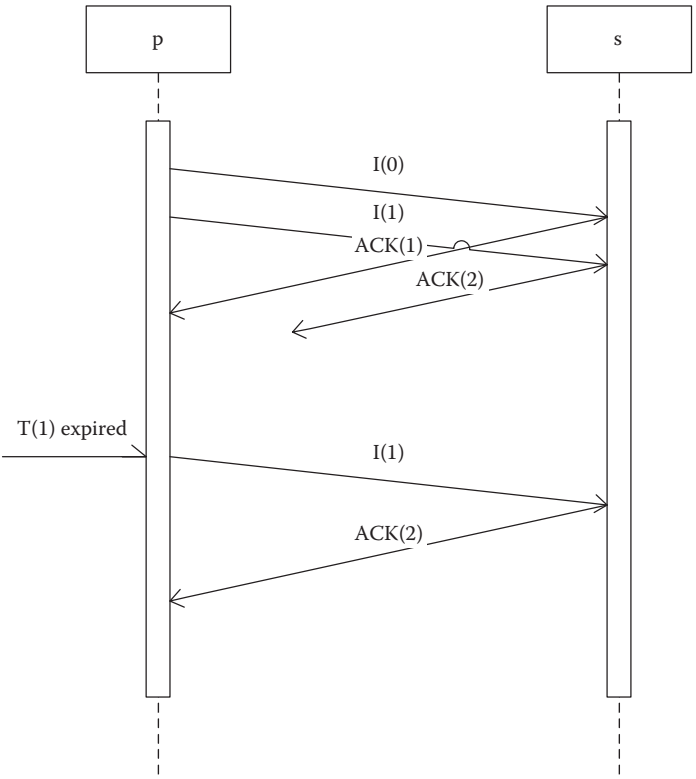


FIGURE 3.70
Example 4 MSC diagram: *I* frame retransmission triggered by the retransmission timer.

3.10.5 Example 5

In this example, we design the SIP INVITE client transaction in accordance with RFC 3261, Section 17.11. First, let us return to the requirements and analysis of a SIP Softphone, introduced as an example at the end of Chapter 2. In that example, we constructed the use case diagram and transformed it into the corresponding general collaboration diagram. At the very end of that example, we showed the one particular collaboration related to the successful session establishment.

Now, let us zoom in on the general collaboration diagram of a SIP Softphone with the focus on the SIP INVITE client transaction and the surrounding objects with which it directly communicates. The resulting general collaboration diagram is shown in Figure 3.71. The SIP INVITE client transaction is modeled as an unnamed object of the class *InClientT* because this object

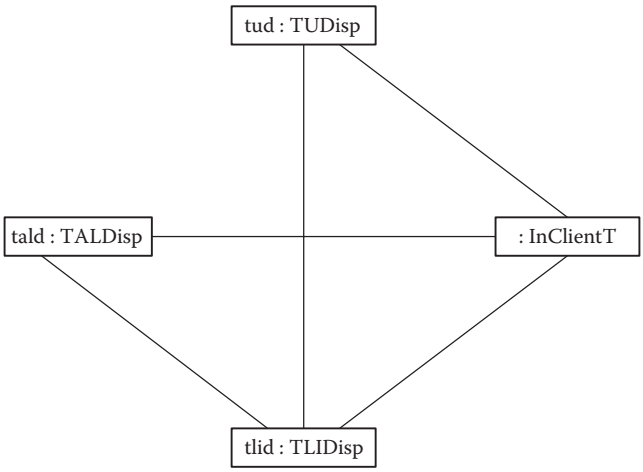


FIGURE 3.71
SIP INVITE client transaction collaboration diagram.

is dynamically created upon user request. It collaborates with the following three objects:

- *tud*, which represents the transaction user dispatcher
- *tald*, which represents the transaction layer dispatcher
- *tlid*, which represents the transport layer dispatcher

Similarly, we can zoom in on the particular collaboration diagram that illustrates a successful session establishment scenario (Figure 2.17) to provide the corresponding particular collaboration of the SIP INVITE client transaction with its surrounding objects (Figure 3.72). As already mentioned in Chapter 2, *req()* and *rsp()* designate requests and responses, respectively. More precisely, *req(INVITE)* is the SIP invite request, *rsp(1xx)* is the SIP provisional response, and *rsp(200)* is the SIP final response. Note that the first message 1:*req(INVITE)* sent from the object *tald* to the SIP INVITE client transaction object in Figure 3.72 corresponds to the fourth message 4:*req(INVITE)* sent from the object *tald* to the SIP INVITE client transaction object in Figure 2.17. Note also that Figure 3.72 shows only the messages exchanged among the objects shown in this figure, and that the sequence numbers of these messages are assigned accordingly.

Another particular collaboration that corresponds to an unsuccessful session establishment scenario is shown in Figure 3.73. This scenario is the same as the previous one up to the step number 6, when instead of the successful final response *rsp(200)*, the unsuccessful final response *rsp(300-699)* is

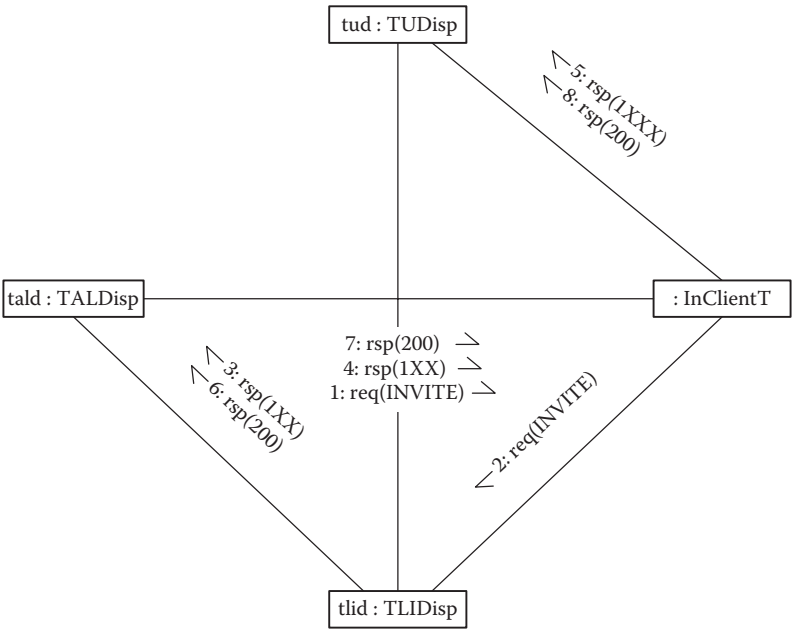


FIGURE 3.72
Successful session establishment collaboration diagram.

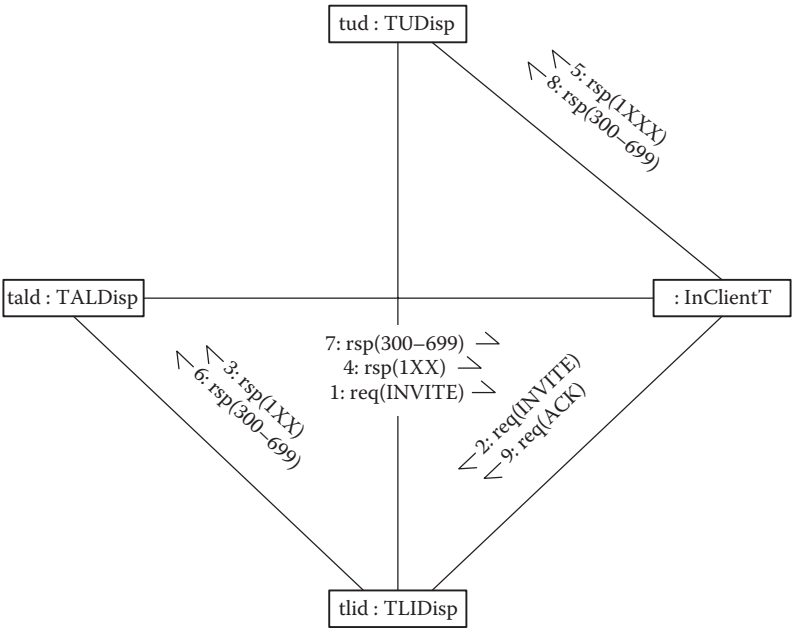


FIGURE 3.73
Unsuccessful session establishment collaboration diagram.

received. In step 7, *tald* forwards *rsp(300-699)* to the SIP INVITE client transaction, which in accordance with RFC 3261, forwards it toward the upper layer and sends the message *ACK* to the remote site. These two actions are performed in steps 8 and 9, respectively. Semantically equivalent sequence diagrams are shown in Figures 3.74 and 3.75. Figure 3.74 illustrates a successful session establishment, whereas Figure 3.75 shows an unsuccessful session establishment scenario.

Based on the SIP INVITE client transaction state transition graph (RFC 3261, page 127) we can construct the corresponding statechart diagram (Figure 3.76). This statechart diagram starts with the transition from the initial state to the state *Calling*, which is triggered by the reception of the signal (message) *req(INVITE)* from the transaction user (TU). The signal *req(INVITE)* models the original request SIP INVITE. During this transition, the SIP INVITE client transaction forwards the message *req(INVITE)* to the transport layer.

At the entrance to the state *Calling*, two timers are started, timer A (TA) and timer B (TB). The former corresponds to the time interval that must elapse before the response to the request INVITE can be received, whereas the latter limits the time interval during which the SIP INVITE client transaction waits for the response to the request INVITE. Initially, TA is set to the value T1 (estimated round-trip time, RTT, which is by default 500 ms) and TB is set to 64 × T1.

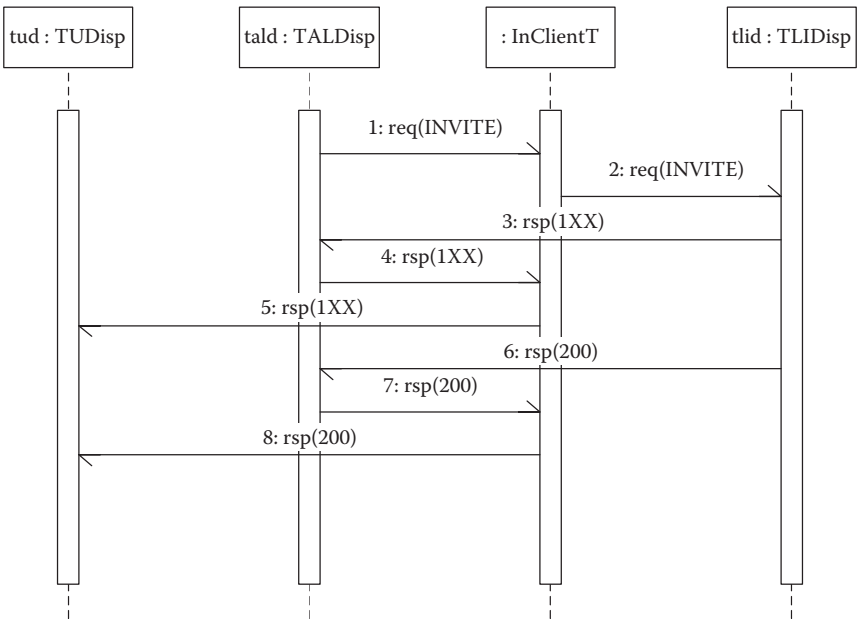


FIGURE 3.74
Successful session establishment sequence diagram.

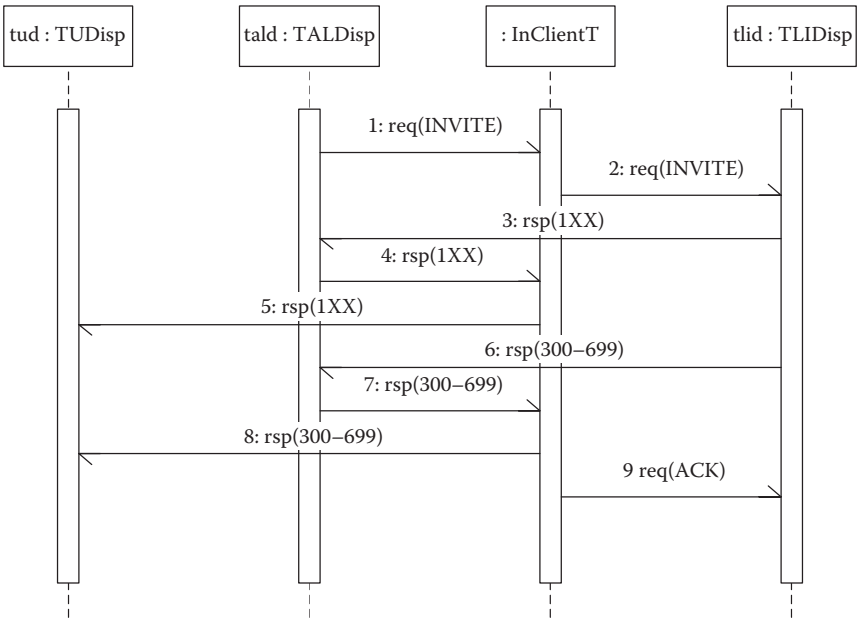


FIGURE 3.75
Unsuccessful session establishment sequence diagram.

If the timer TA expires, the SIP INVITE client transaction restarts it by doubling its current value (TA = TA × 2) and retransmits the signal *req(INVITE)*. Initial values of TA and TB (T1 and 64 × T1, respectively) allow this procedure to repeat the maximum of seven times before the timer TB expires. If the timer TB expires (or if a transport error is detected), the SIP INVITE client transaction informs TU accordingly and moves to the state *Terminated*, and from there to its final state.

Most frequently, a response to the request INVITE will be received before the timer B expires. In such a case, the SIP INVITE client transaction stops both timers and moves to the next state, which depends on the type of response. If the provisional response *rsp(1xx)* is received, the SIP INVITE client transaction forwards it to TU and moves to the state *Proceeding*. If the successful final response *rsp(2xx)* is received, the SIP INVITE client transaction forwards it to TU and moves to the state *Terminated*. If the unsuccessful final response *rsp(300-699)* is received, the SIP INVITE client transaction forwards it to TU and sends the signal (message) *ACK* to the remote site.

While being in the state *Proceeding*, the SIP INVITE client transaction simply forwards all the preliminary responses *rsp(1xx)* to TU. Once it receives the successful final response *rsp(2xx)*, it also forwards it to TU and moves to the state *Terminated*. If the SIP INVITE client transaction receives the

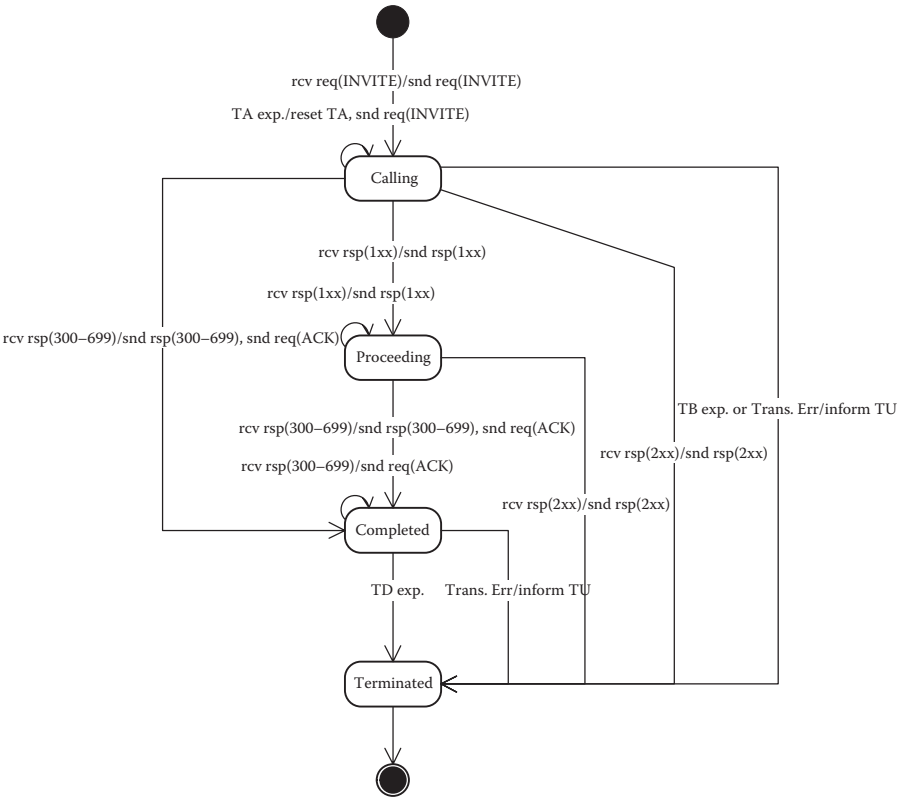


FIGURE 3.76
Statechart diagram of the SIP INVITE client transaction.

unsuccessful final response *rsp(300–699)* in the state *Proceeding*, it forwards that response to TU, sends the signal *req(ACK)* to the remote site, and moves to the state *Completed*.

At the entrance to the state *Completed*, the third timer, the timer D (TD), is started. While being in the state *Completed*, the SIP INVITE client transaction just confirms any unsuccessful final responses *rsp(300-699)* by sending the SIP message *ACK* to the remote site. If the SIP INVITE client transaction detects a transport error, it informs TU accordingly and moves to the state *Terminated*. Finally, when the timer D expires, the SIP INVITE client transaction finishes simply by moving to the state *Terminated*.

We finalize this example with the semantically equivalent SDL diagram, which, due to its size, is shown in the next four figures (in these figures, TPL stands for the transport layer and TU stands for the transaction user). Figures 3.77 through 3.80 illustrate the processing of events in the states *Calling*, *Proceeding*, *Completed*, and *Terminated*, respectively.

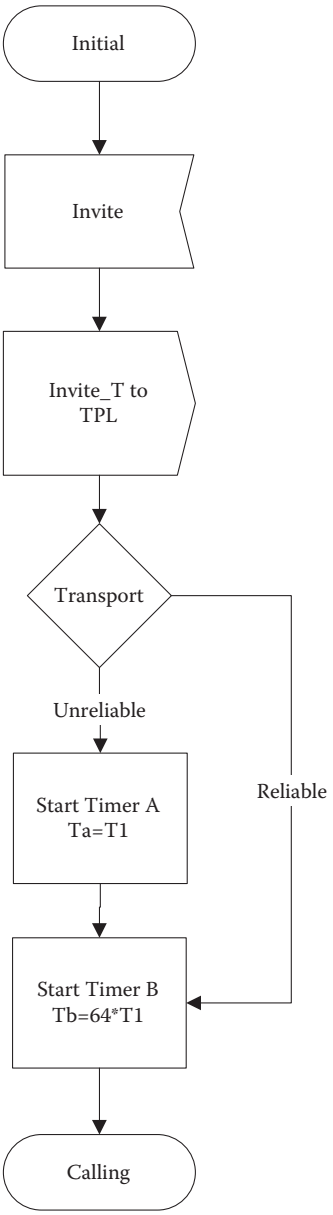


FIGURE 3.77
SDL diagram of the SIP INVITE client transaction, part I.

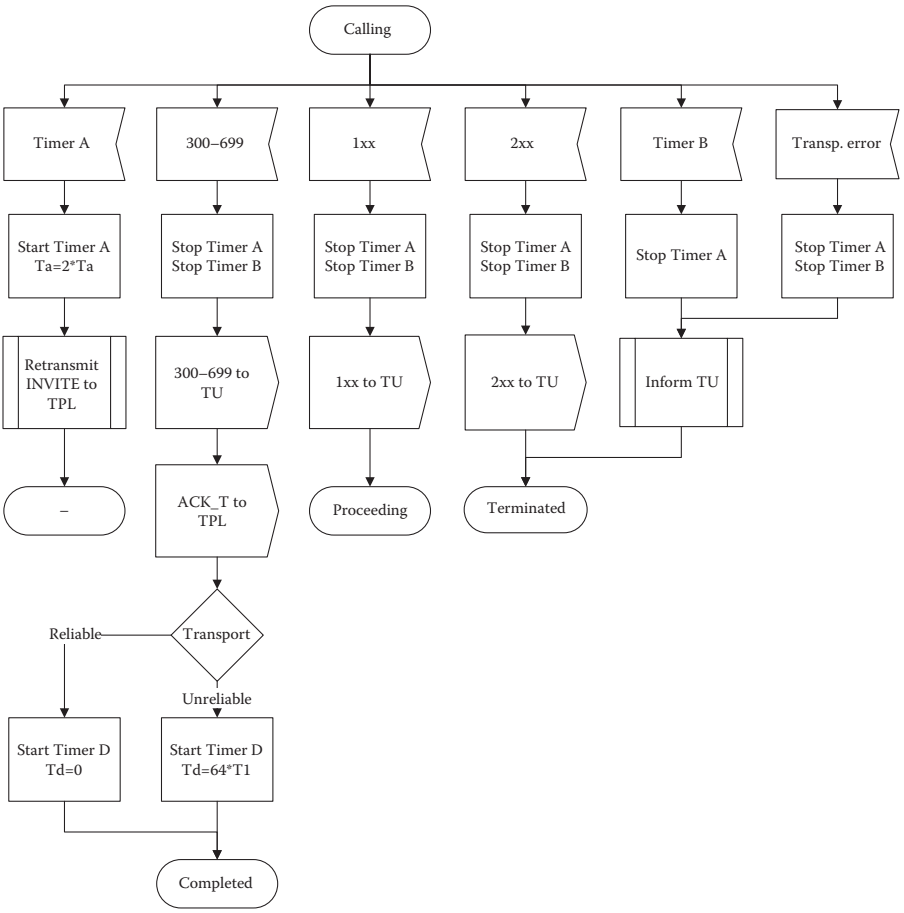


FIGURE 3.78
SDL diagram of the SIP INVITE client transaction, part II.

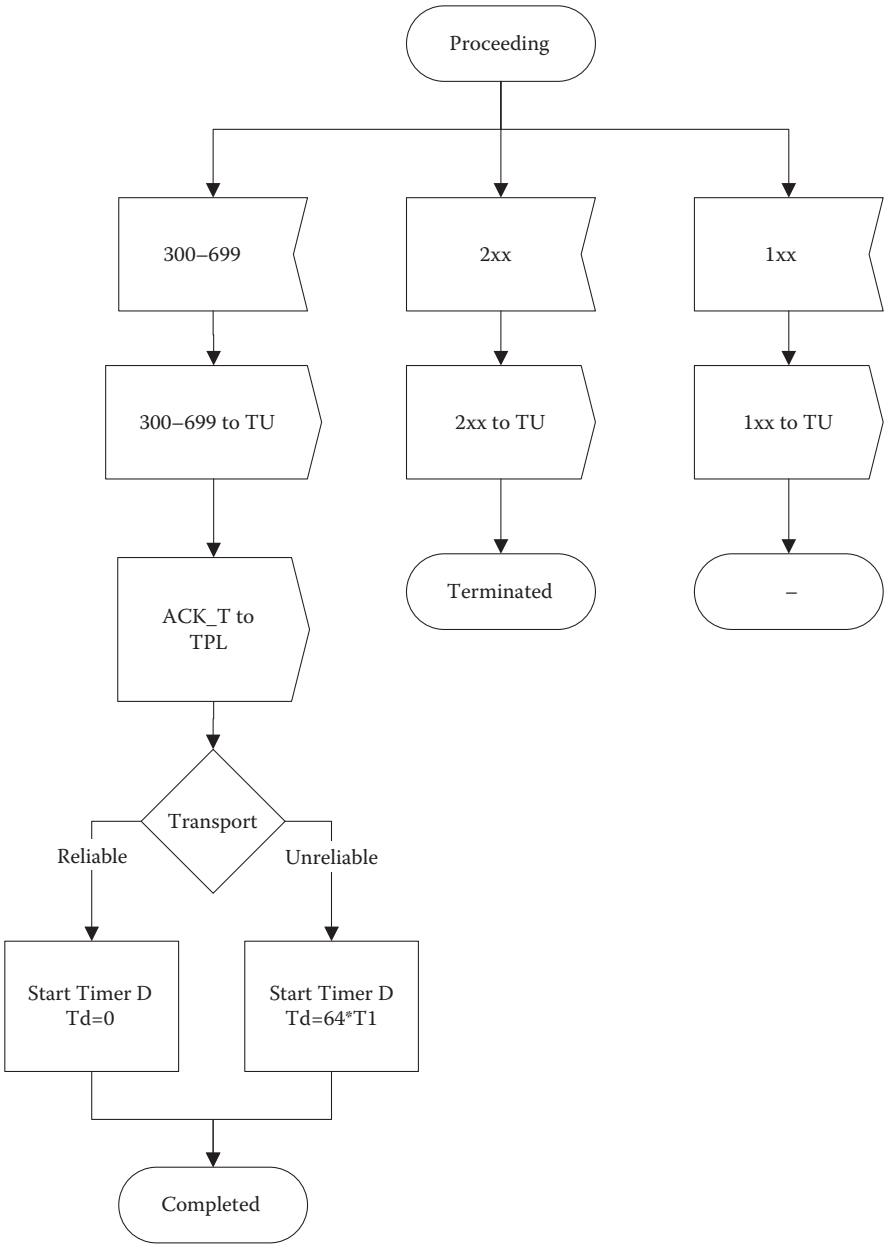


FIGURE 3.79
SDL diagram of the SIP INVITE client transaction, part III.

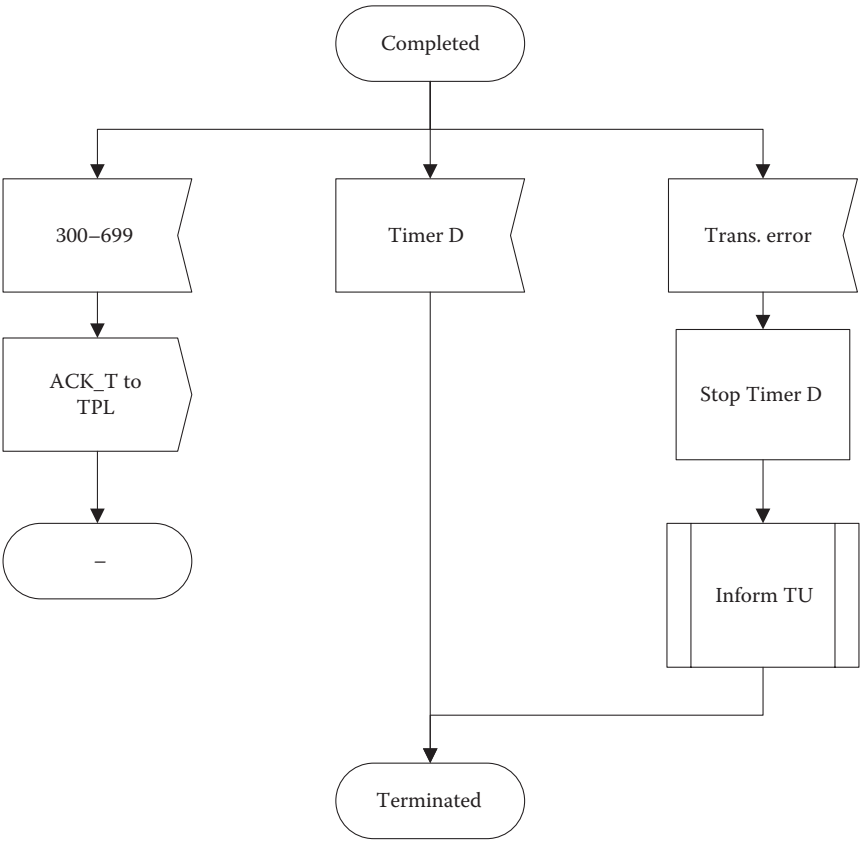


FIGURE 3.80
SDL diagram of the SIP INVITE client transaction, part IV.

References

Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1998.

Willock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., Schulz, S., *An Introduction to TTCN-3, Second Edition*, John Wiley & Sons, Chichester, West Sussex, UK, 2011.

Halsall, F., *Data Communications, Computer Networks and OSI*, Addison-Wesley, Reading, MA, 1988.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

4

Implementation

The system **implementation** is a phase in engineering work that follows the system design phase. This phase consists of the following two steps:

- Transform a design model into the implementation model
- Transform the implementation model into a higher-level programming language code

A design model is given in the form of the corresponding UML (Booch et al., 1998) or SDL diagrams, which are the results of the previous phases of communication protocol engineering, i.e., requirements, analysis, and design. The implementation model takes the form of the corresponding UML component diagram. The output of the implementation phase is a set of source code modules, today most frequently in C/C++ or Java, which is also referred to as the implementation. This may sound confusing, but in reality, the correct meaning of the term is easily deduced from its context.

Logically, *implementation as a phase* of the production process is a well-defined mapping of a design model into a higher-level programming language source code. *Implementation as a product* is a result of this mapping. The attribute *well-defined* reflects the assumption that both detailed procedures and adequate tools are provided for transforming models into program source code. This well-defined mapping of a model into the program source code is referred to as **forward engineering** in UML terminology. Likewise, the reverse mapping of a program source code into the model is referred to as **backward engineering**.

In a mathematical sense, both the mapping of a program into the program source code and the result of that mapping (i.e., the implementation in both of its meanings) are not unique. Therefore, logically, more than one *correct implementation* exists for a given model of the communication protocol. Under the correct implementation, we assume an implementation that for given input produces expected outputs within the expected time frame, which is defined with the corresponding timers. We say for such implementation that it is compliant (conformant) with (to) the given model. The terms *compliant* and *conformant* are synonyms in this context. If the model has been standardized (e.g., by IETF or ITU-T), we say that the implementation is compliant with the standard.

The concept of forward and backward engineering is an intriguing one. Proponents of the model-based software development and various initiatives in Model-Driven Architecture (MDA) strongly believe that forward and backward engineering is possible, and they are putting forth tremendous efforts to make it real. Quite a number of commercially available tools are made with this goal in mind. The agile programming community is strongly opposed to it because their members believe that only the program source code is the complete specification of the system. From their point of view, only the set of test cases that successfully pass are proof that the implementation is correct.

Other groups also exist between these two extremes that are trying to close the gap between software modeling and programming (also called coding). For example, the creators of the StateWORKS® tool and the corresponding approach claim that although UML tool vendors made serious attempts to generate code from models, they are facing major difficulties, and that these tools can so far produce only header files or code skeletons. As an alternative, they introduced the notion of the totally complete models in an attempt to completely eliminate programming. The models in StateWORKS® are sets of virtual finite state machines (VFSMs) that run on top of the VFSM Executor, which is essentially an interpreter.

This book has a similar but different approach. We try to shrink the gap between communication protocol modeling and programming, both by making detailed models and by providing the FSM library, which forces programmers to transform models into code in a uniform way. This methodology makes forward engineering well defined. As already mentioned in the previous chapter, the FSM library provides two main classes, namely *FiniteStateMachine* and *FSMSystem*. The former is used to model and implement individual FSMs and the latter is used as their execution platform, which comprises common services and an event (message) interpreter.

When it comes to programming interpreters and FSM-related libraries, a broad spectrum of possible implementations exists, starting with the traditional structural or procedural solution, continuing with a series of mixed solutions, and ending with the object-oriented solutions of both static and dynamic type. This situation is justified by the fact that the implementation style depends highly on the type of target architecture. For example, if we consider a microcontroller as the target architecture, we are naturally forced to select a structural solution in the C/C++ programming language. If we consider more powerful architectures, in terms of resources, we may also take into consideration the object-oriented approaches supported by the C++ and Java programming languages.

In Section 4.1, we introduce the component diagrams, which are the means of making implementation models. We then illustrate a spectrum of possible finite state machine implementations, including the catalogued state design

pattern (Gamma et al. 1995), which is explained in Section 4.3. After that, we cover the concepts and, most importantly, the design and implementation details of the FSM library (its reference manual is given in Chapter 6). We conclude this chapter with two implementation examples.

4.1 Component Diagrams

In Chapter 3 we were dealing with abstractions in the conceptual world. The design phase typically starts with exploration in the realm of interaction diagrams, where we try to get a better feeling of the system. We finish the design phase by defining the static structure and the complete behavior of the system in the corresponding class and activity, or statechart diagrams, respectively. At the end of the design phase, we also specify the deployment of individual software components by rendering the corresponding deployment diagrams.

In the implementation phase, we are materializing the design abstractions (such as classes, interfaces, and collaborations) into the components that live in the physical world. As already mentioned, a component is a physical and replaceable part of the system that realizes the given set of interfaces. What we actually do at the beginning of the implementation phase is pack the design abstractions into packages with well-defined interfaces, referred to as components. Examples of such packages are traditional binary object libraries, dynamically linkable libraries (DLLs), and executables; as well as tables, files, and documents.

The components and classes are very much alike. Both can:

- Realize a set of interfaces
- Participate in relations (dependencies, generalizations, and associations)
- Be nested
- Have instances
- Participate in interactions

The differences between the components and the classes are as follows:

- The former represents physical entities, whereas the latter is a conceptual abstraction, so they exist on different levels of abstraction.
- The former only has operations that are accessible through its interfaces, whereas the latter may have both operations and attributes.

The most important feature of the component is that it is replaceable. This means that we can substitute a component with another one without

any influence on the system as a whole. This replacement is completely transparent to the users of the replaced component. A new component provides the same or perhaps even better services through the exact same interfaces.

We distinguish the following three types of components:

- The deployment components (already introduced in the context of deployment diagrams) are the parts of the executable system, such as executables and DLLs.
- The work product components are the artifacts of the development process (such as project settings or the source code) and data files that are used to build the deployment components.
- The executable components are the parts of the run-time system, e.g., DCOM and CORBA components.

We make the implementation models by rendering the component diagrams. The set of graphical symbols that are available for rendering component diagrams is shown in Figure 4.1. As usual, we select a symbol from the set of available symbols, drag and drop it onto the working sheet, and fill in the data related to its properties. The set of symbols available for rendering component diagrams is obviously a subset of the set of symbols available for rendering deployment diagrams. The properties of these symbols are explained in Chapter 3 (see Section 3.6).

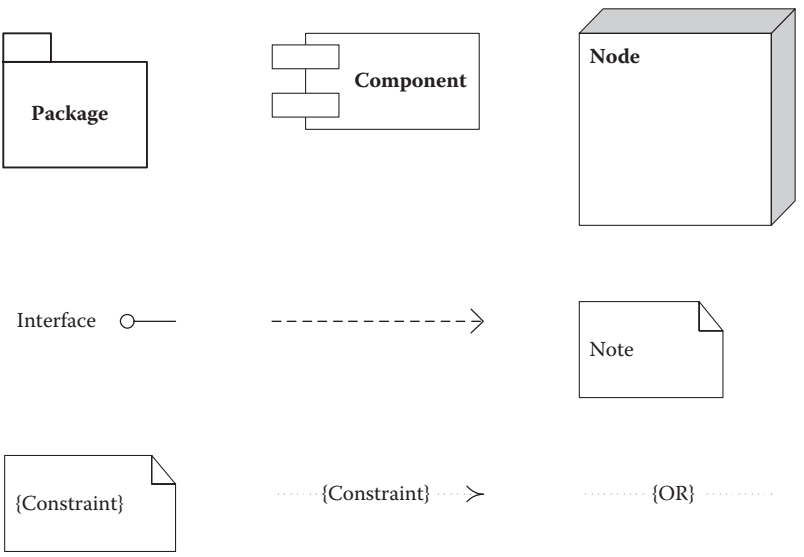


FIGURE 4.1
Set of symbols available for rendering component diagrams.

In communication protocol engineering, we are mainly using component diagrams for:

- Modeling APIs
- Modeling executables and libraries
- Modeling source code

Well-defined application programming interfaces (APIs) are some of the most important features of the well-structured software system. An API is an interface that is realized by one or more components. Being an interface, it actually defines a set of services. It represents a clear demarcation line between the service users and the service providers. The former receives the service without caring who is providing it. The same also holds true in the opposite direction: the latter provides the service without caring who receives it.

We may think of APIs as the programmatic seams of the system. We use them to connect more components together to create more complex systems. Each component is replaceable. We can replace it with another component whenever there is a need. The developers of the component that use some APIs do not care who or how it will be provided. They only care about how to fulfill the requirements for the component they are working on currently. Alternately, the system integrator must care that all of the needed components are provided and that they are compliant with their APIs.

Figure 4.2 illustrates the modeling of APIs by means of a very simple example. Imagine that we have been provided with the TCP/IP protocol stack packed as a dynamically linkable library, named *tcpipstack.dll*. It defines the API that comprises three interfaces, namely, *TCPSockets*, *UDPSockets*, and *IPInterface*. The first provides communication services over TCP ports, the second over UDP ports, and the third directly over IP.

Provided with such a component, we are now able to create a new component that uses it. For example, we can create the DLL *sip.dll* (Figure 4.3). This

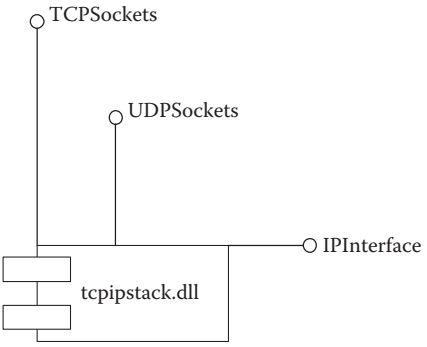


FIGURE 4.2
Example of a simple API.

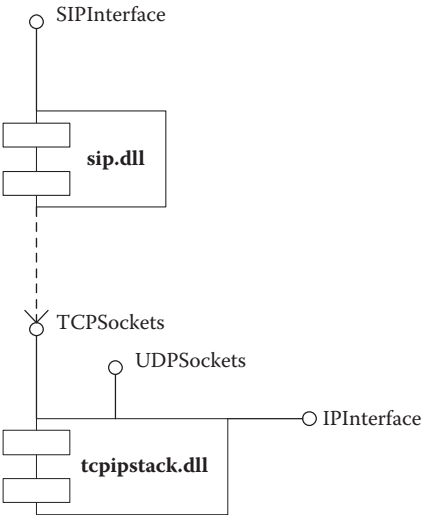


FIGURE 4.3
Example of a simple API user.

new component provides the SIP services through the interface *SIPInterface*. The fact that *sip.dll* uses services provided through the interface *TCPSockets* is modeled by connecting these two with the dependency relation.

Besides modeling APIs, we can use component diagrams to model executables and libraries. Generally, if the system under development comprises more executables and associated object libraries, it may be wise to make a model that illustrates their relationships. This is especially important if we want to keep versioning and configuration management during the system lifetime under control.

Modeling of executables and libraries can help in making the decision regarding the physical partitioning of the system. The issues that affect this decision-making are as follows:

- Technical issues
- Configuration management issues
- Reusability issues

Figure 4.4 shows the model of a simple executable, named *softphone.exe*. This executable uses the DLL *sip.dll* through the API that comprises the single interface *SIPInterface*. Farther down the hierarchy, *sip.dll* receives the communication service that is provided by the DLL *tcpipstack.dll* through the interface *TCPSockets*.

Each library and executable is built in the environment of a separate software project. Generally, a software project comprises the project configuration (settings) files, the source code files, and the object libraries. The source

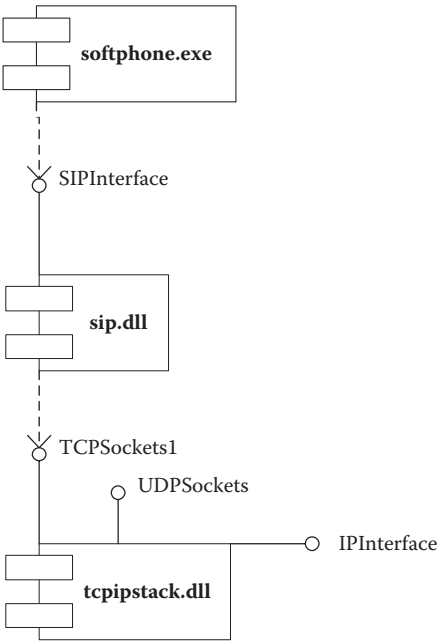


FIGURE 4.4
Model of a simple executable.

code files typically include the module declaration (header) and the module definition files. The developers try to logically organize these files into a file system structure by placing the related files into the same directory (folder).

In the case of complex projects, the corresponding directory tree can get rather ramified, and sometimes it may not be clear where to put new software modules. This can be especially confusing for the new members of the development team. Things get even worse when we must manage splitting and merging of groups of files as development paths fork and join.

In such cases, it is advisable to make a model of the software project, also referred to as the source code model. An example of such a model is shown in Figure 4.5. The executable *Main.exe* is built in accordance with the project definition file *Main.dsw*. Because the project comprises all the module headers and module definition files, the file *Main.dsw* has a dependency relation with all of them. (For clarity, only some of these dependencies are shown in Figure 4.5.)

Farther down the hierarchy, the source code files *AutomataA.cpp* and *AutomataB.cpp* use the header files *AutomataA.h* and *AutomataB.h*, respectively. Both of these header files use the header file *Constants.h*. Finally, all of the header and source code files, except *Constants.h*, use the framework *FSMLibrary*.

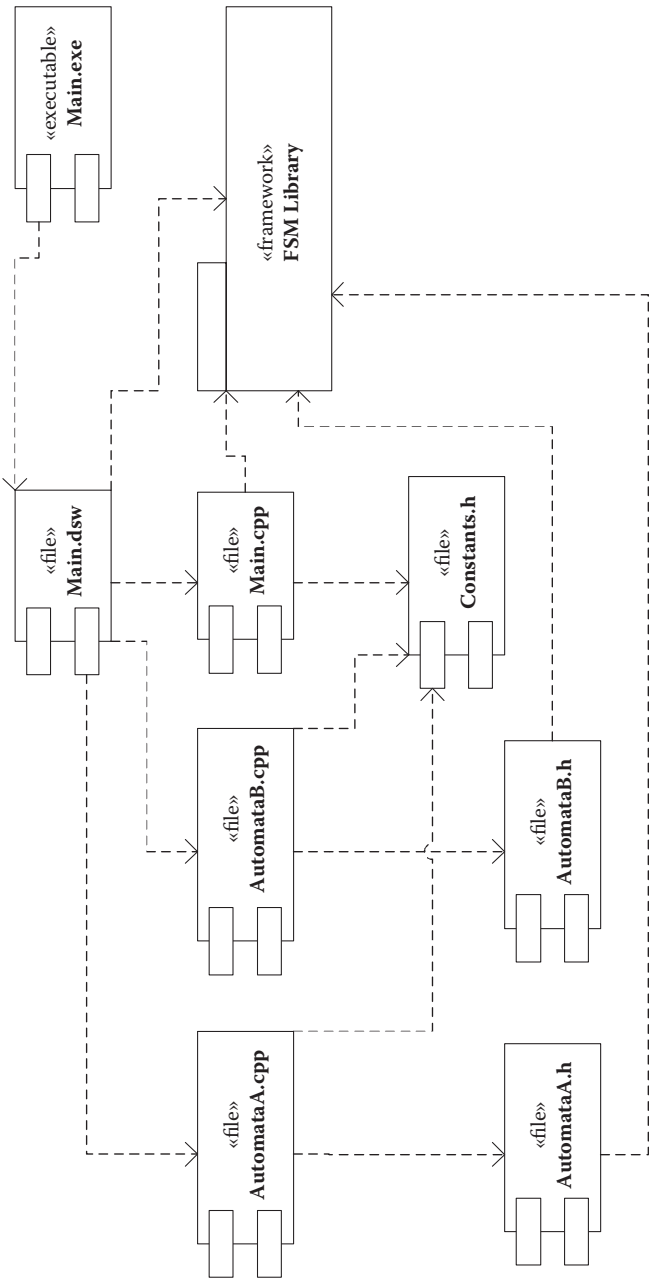


FIGURE 4.5
Model of a simple project.

4.2 Spectrum of FSM Implementations

As mentioned in Chapter 3, we model communication protocols as finite state machines (FSMs). A broad spectrum of various solutions exists for the implementation of FSMs. This section contains a short overview of only three, perhaps the most representative approaches to the implementation of FSMs. The complete treatment of all methodologies and corresponding tools is outside the scope of this book, and as an alternative we simply want to develop ideas by exploring different implementations of a simple FSM (counter by modulo 2). The goal is to familiarize the reader with this subject by showing what the problems are and how they can be tackled.

The three approaches to FSM implementation are illustrated by simple implementations of modulo 2 counters in the Java programming language. As already mentioned, communication protocol developers today mainly use C/C++ and Java, and the selection of the programming languages for certain projects mainly depends on the target platform. By mixing examples in Java and C/C++, we want to show that all these languages are applicable in the area of communication protocol engineering, and that the selection of a programming language is not the highest priority issue. Actually, we start with Java in Sections 4.2 and 4.3, switching to C++ later.

The state design pattern is a particular FSM implementation type that is special because it was catalogued by Gamma et al. in 1995. Because of that, it receives its own separate section. However, none of these four approaches are used later in this book. Instead, we introduce the FSM Library-based implementation paradigm, which is more like the state-of-the-art paradigm. In other words, first we show what is possible, and perhaps what is next, and then we turn to the current practice in communication protocol engineering.

Let us turn our attention to the subject of the implementation, a communication protocol. As already mentioned in Chapter 1, the communication protocol is defined with the syntax of its messages, the set of procedures (actions) that process the messages, and the set of reactions to exceptional events (timer and error management). In the programming world, they are modeled as finite state machines, also referred to as automata. Mathematically, the abstract automata are defined as

$$A = (X, Y, S, t, o, S_0)$$

where

$X = \{X_1, X_2, \ldots X_n\}$ is a set of input signals (input alphabet)
$Y = \{Y_1, Y_2, \ldots Y_m\}$ is a set of output signals (output alphabet)
$S = \{S_1, S_2, \ldots S_k\}$ is a set of states (state alphabet)
S_0 is the initial state
t is the transition function that maps the Cartesian product of S x X to S
o is an output function that maps the Cartesian product of S x X to Y

Abstract automata are typically illustrated in the form of a state transition graph. The example of the state transition graph in Figure 4.6 illustrates the counter by modulo 2, which is actually the example of a finite state machine we want to implement in Java. It is formally defined as follows:

$$C = (X, Y, S, t, o, S_0)$$

where

X = {0, 1}
Y = {0, 1, 2}
S = {*S1*, *S2*, *S3*}
S_0 = *S1*

The functions *t* and *o* are defined in Table 4.1.

The input and output alphabets comprise the signals {0, 1} and the signals {0, 1, 2}, respectively. The automata can take one of the three possible states, namely, *S1*, *S2*, and *S3*. The initial state of the automata (S_0) is the state

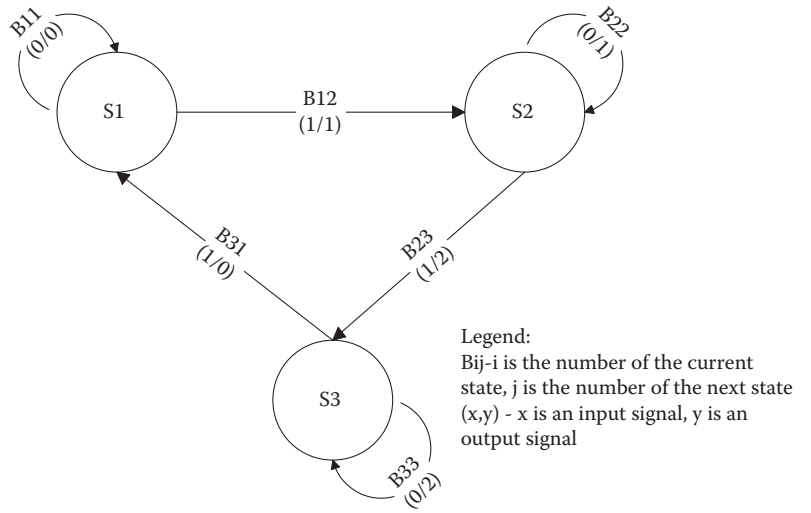


FIGURE 4.6
Counter by modulo 2 state transition graph.

TABLE 4.1

The Counter by Modulo 2 Transition Table

Next State//Output Signal	Input Signal 0	Input Signal 1
State *S1*	1/0	2/1
State *S2*	2/1	3/2
State *S3*	3/2	1/0

S1. Both transition and output functions are defined in Table 4.1. The rows of this table correspond to the automata states (*S1*, *S2*, and *S3*), whereas the columns correspond to the input signals (0 and 1). The elements of Table 4.1 have the format *s/y*, where *s* corresponds to the next state number and *y* corresponds to the output signal.

The same information about the next state and the output signal is shown differently in the state transition graph (Figure 4.6). The arcs of the state transition graph are labeled as $B_{ij}(x/y)$, where *i* is the number of the current state, *j* is the number of the next state, *x* is the input signal that triggers the transition, and *y* is the output signal generated by the transition. The corresponding statechart diagram is shown in Figure 4.7.

The simplest but perhaps still the most frequently used FSM implementation is based on the structural or procedural approach. This implementation is made in the form of nested selection statements in higher-level programming languages. In the programming languages C/C++ and Java, we typically use *switch–case* statements for this purpose, because the control flow structures made with *if* and *else–if* statements are less readable.

Typically, the outermost *switch–case* statement selects a case that corresponds to the current state of automata. In the code paragraph that defines the processing of the current state, normally we use the second, nested *switch–case* statement, which selects the case that corresponds to the input signal. The program paragraph that corresponds to that input signal effectively performs the transition by creating the corresponding output signals and evolving to the next state. This evolution is made simply by updating the content of a variable that holds the identification of the current state (most frequently, this is just the index of the state).

Actually, the structure of the resulting program code is very similar to the program representation of SDL (SDL-PR), which was introduced in Chapter 3, and this fact was also mentioned there. Generally, communication

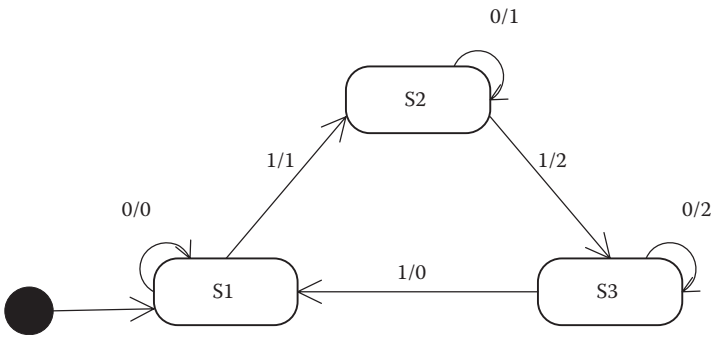


FIGURE 4.7
Counter by modulo 2 statechart diagram.

protocol implementation based on nested *switch–case* statements looks like the following:

```
switch(state) {
 case STATE_1:
  switch(message_code) {
   case MESSAGE_CODE_1:
    // processing of the message code 1 in the state 1
    break;
   case MESSAGE_CODE_2:
    // processing of the message code 2 in the state 1
    break;
   case MESSAGE_CODE_3:
    // processing of the message code 3 in the state 1
    break;
    ...
    default:
     // processing of the unexpected message in the state 1
     break;
  }
 case STATE_2:
  switch(message_code) {
   case MESSAGE_CODE_1:
    // processing of the message code 1 in the state 2
    break;
   case MESSAGE_CODE_2:
    // processing of the message code 2 in the state 2
    break;
   ase MESSAGE_CODE_3:
    // processing of the message code 3 in the state 2
    break;
   ...
   default:
    // processing of the unexpected message in the state 2
    break;
  }
 ...
 case STATE_N:
...
}
```

We illustrate this general scheme by applying it to the implementation of the counter by modulo 2 in Java. The three states of the counter are labeled as *S1*, *S2*, and *S3* in the program code. The input signals 0 and 1 are labeled as *M1* and *M2*, respectively. The demonstration program reads the actual input signals from the standard input file (by default, this is the keyboard). The generated output signal is represented by a simple printout on the standard output file (by default, this is the monitor). The demo program code is the following:

```
package automata;
import java.util.*;
import java.io.*;
public class Environment1 {
 public static void main(String[] args) throws IOException {
  char ch = '0';
  Automata1 a1 = new Automata1();
  System.out.println("This is the example of counter by modulo 2.");
  System.out.println("Automata evolution has started...");
```

```
  while(true) {
   System.out.print("Enter input signal (0/1 and <ENTER>):");
   ch = (char)System.in.read();
   System.in.skip(2);
   if(((ch!='0') && (ch!='1'))) break;
   a1.processMsg(ch);
  }
 }
}
```

The demo program initially creates the object *a1*, an instance of the class *Automata1*, which is the structural and procedural implementation of the counter by modulo 2. After printing two welcome messages, it falls into an infinite *while* loop in which it prompts the user for the input signal and reads it. If the input signal is neither 0 nor 1, the demo program breaks the loop and terminates. Otherwise, it performs one step of the automata evolution by calling the procedure *processMsg()* of the object *a1*.

The Java code for the class *Automata1* is the following:

```
package automata;
public class Automata1 {
 private static final int S1 = 0;
 private static final int S2 = 1;
 private static final int S3 = 2;
 private static final char M1 = '0';
 private static final char M2 = '1';
 private int state=S1;
 public void processMsg(char msg) {
  switch(state) {
   case S1:
    switch(msg) {
     case M1:
      System.out.println("Output signal: 0");
      break;
     case M2:
      System.out.println("Output signal: 1");
      state = S2;
      break;
     default:
      break;
    }
   break;
 case S2:
  switch(msg) {
   case M1:
    System.out.println("Output signal: 1");
    break;
   case M2:
    System.out.println("Output signal: 2");
    state = S3;
    break;
   default:
    break;
  }
  break;
 case S3:
  switch(msg) {
   case M1:
    System.out.println("Output signal: 2");
    break;
```

```
    case M2:
     System.out.println("Output signal: 0");
     state = S1;
     break;
    default:
     break;
    }
    break;
   default:
   break;
   }
  }
}
```

The implementation above starts with the definition of the symbolic constants that correspond to the possible automata states (namely *S1*, *S2*, and *S3*) and valid input signals *M1* and *M2* (input signals 0 and 1). Next, we define the variable state that holds the current automata state and we set it to the value *S1* (the automata initial state).

The method *processMsg* starts with the *switch–case* statement that selects the further execution path depending on the content of the variable state (i.e., the current automata state). Three possible cases are found that are defined by the corresponding case clauses. Each of these clauses contains a further *switch–case* statement that distinguishes between two valid input signals, namely *M1* and *M2*. The nested *case* clause that corresponds to the particular input signal prints the message, which corresponds to the output signal, and updates the variable *state*, if the current state of the automata changes.

This example demonstrates the main advantage of the structural or procedural approach: simplicity, which yields greater performance in terms of execution speed. Another advantage is that we can easily construct a compiler or a code generator that generates such implementations (a good example that justifies this claim is SDL-PR). The main disadvantage of this approach is its bad scalability, which becomes evident in the case of large-scale implementations, i.e., implementations of automata that have a large number of states and state transitions.

The code size for such program implementations increases linearly with the number of states and the number of state transitions. Another disadvantage of this approach is that it is monolithic which implies that it is static regarding the possible need to change the automata, either by adding new, or deleting the existing states, or by adding or deleting state transitions.

In this type of implementation, the structure of the automata (its vertex and arcs) is built into the machine code of the implementation (hard-coded). We say that the input signal processing flow is governed by the structure of the machine code. If we want to add or delete a state or a state transition, we must change the program code, recompile it, and install the new version on the target platform. Most frequently, the installation procedure requires the system to be restarted at its end. Restarting the system means that effectively it will not be operational for a certain short interval of time. The problem

is that some types of systems, such as nonstop systems, may not tolerate restarts no matter how short the time interval is.

Some systems try to make restarts allowable by providing processor tandem configurations. Typically, in such a system, one of the processors continues the normal operation while the other restarts after an update. In that case, we have a synchronization problem, which of course can be solved but could become rather complex. Generally, system restarts are problematic and should be handled with special care.

On the other end of the spectrum of FSM implementations, we have the diametrical approach to FSM implementation in which the structure of the automata is not defined by the program control flow, but rather with the corresponding data structure. The simple interpreter uses this data structure to process the incoming events (messages), therefore it is referred to as an event interpreter. The data structure implementations in assembler and C programming languages are built from lists and lookup tables.

The automata evolution is driven by the incoming events. Each input event triggers one step of the evolution. The event interpreter carries out the evolution step by traversing the data structure to determine the current state and the state transition that corresponds to the input event type. In contrast to this common part of the message processing flow—which is directed by the data structure—program parts that correspond to particular reaction tasks are dedicated routines that perform specific functions, which cannot be generalized.

Figure 4.8 illustrates the FSM implementation based on the event interpreter and the data structure that defines the FSM structure (essentially, the state transition graph). New, incoming events (messages) are added at the end of the message queue (see the top left corner of Figure 4.8). The interpreter takes the messages from the head of the message queue and processes them by using the data structure, which comprises

- An automata control table
- An automata state table
- A list of valid events (one such list exists for each automata state)

The automata control table is assigned to automata to store its current state and optionally some of its additional attributes. The automata state table is a lookup table that maps the state index onto the address of the corresponding list of valid events in that state. The elements of this list contain the complete information necessary and sufficient to perform the state transition from the current state to the next state, which is determined by the event type. This information is stored in the following fields:

- *event ID*: holds the event type to which this element corresponds
- *task address*: contains the pointer to the corresponding routine (procedure)

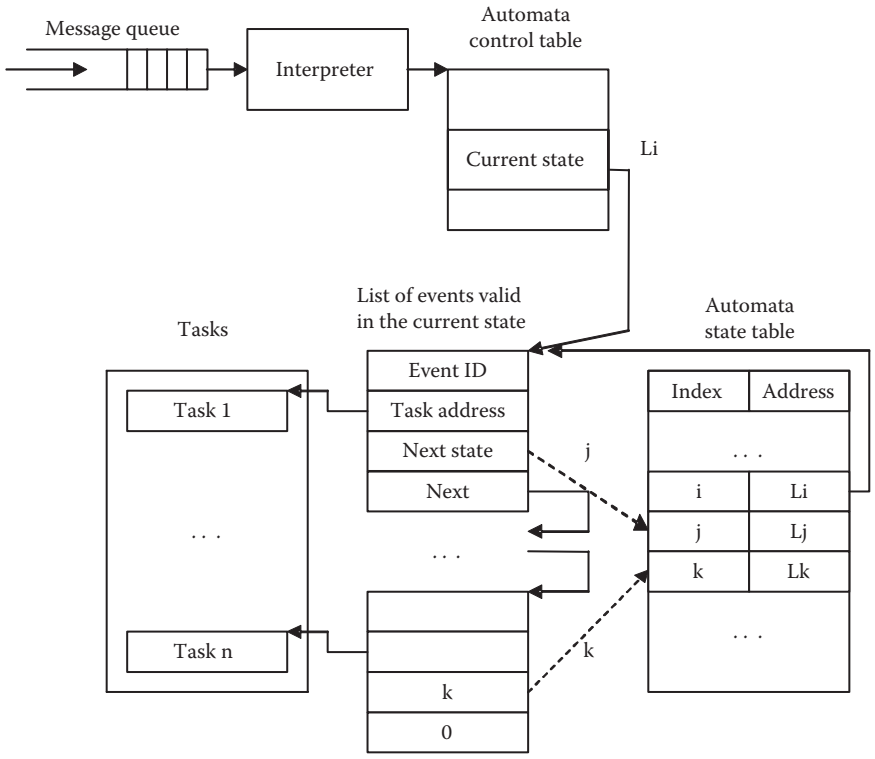


FIGURE 4.8
Event interpreter and the data structure that defines the FSM structure.

- *next state*: stores the index of the next state
- *next*: contains the pointer to the next element in the list

The event interpreter processes the message through the following steps:

- Get the message from the head of the message queue.
- Locate the automata control table by examining the content of the message header (the message destination field, in particular).
- Read the current state and locate the corresponding list of valid events by looking up the automata state table.
- Determine the event type by examining the content of the message header (the message code field, in particular) and locate the corresponding element in the list of valid events (ignore the event if such an element does not exist).
- Perform the task by calling the corresponding task routine as a subroutine (procedure).

- Read the index of the next state from the field *next state*.
- Update the field *current state* by storing the index of the next state into the field *current state*.

The advantage of this approach is that we can construct a compiler that transforms the design FSM model into the corresponding data structure and the set of task routines. The automatic translation performed by the compiler increases the probability that the implementation is compliant with the design model and, therefore, that it is correct. Moreover, the routine performed by the event (described above) is fairly simple and short. The price that is paid for the correctness and simplicity is poor performance. The decrease in the processing throughput is proportional to the number of memory accesses to the corresponding elements of the data structure.

Two characteristics of this approach are not obvious from Figure 4.8 and require further explanation. The first characteristic is universality. Since the FSM structure is built into the corresponding data structure, the event interpreter routine is completely independent from it. The event interpreter always repeats the same routine. This is the same for all FSMs. Therefore, this routine is universal in contrast to the implementation with nested **switch–case** statements, which implement just one particular FSM. This characteristic is especially important from the point of software maintenance. If we want to change the FSM structure by adding or deleting states or state transitions, we must update the data structure. There is no need to change the simple interpreter routine at all.

The second characteristic of the event interpreter-based approach is that it enables sharing of common tasks between more state transitions. In principle, this is also possible in the nested **switch–case**-based approach by introducing common functions, which are called from the corresponding case program clauses, but this is seldom used by their practitioners. In the event interpreter-based approach, this possibility becomes more apparent and is, therefore, implemented because tasks are already specified as procedures (subroutines) rather than *case* program clauses.

Because of task sharing, the number of tasks may generally be smaller than the number of state transitions. We can also organize tasks hierarchically, such that higher-level tasks call their subordinate tasks. This makes it possible to implement more complex tasks by using simple primitives. Such organization has the following advantages:

- Better performance in terms of code size
- Enables dynamic mutation of tasks

By exploiting these characteristics in environments with dynamic loaders, such as Java, we can implement dynamically reconfigurable automata. The automata in such environments change during normal system operation, and those

changes do not demand any system restarts. In such environments, it is desirable to use the object-oriented approach and to define the FSM structure with the set of objects rather than with a data structure, such as the one previously described. The event interpreters in such implementations interact with the objects that materialize the FSM structure instead of using the traditional data structures.

The following code illustrates FSM structure modeling with the group of classes written in Java:

```
package automata2;
import java.util.*;
import java.io.*;

class Task {
 public int id;
 public Task(int ident) {id=ident;}
 public void processMsg() {System.out.println(id);}
}

class Branch {
 private String msgcode;
 private Task task;
 private String nextstateid;

 public Branch(String msg, Task tsk, String nextsts) {
  msgcode=msg;
  task=tsk;
  nextstateid=nextsts;
 }
 public String getMsgCode() {return msgcode;}
 public Task getTask() {return task;}
 public String getNextStateId() {return nextstateid;}
}

class State {
 private String stateid;
 public Set setofbranches;

 public State(String id,Set branches) {
  stateid=id;
  setofbranches=branches;
 }
 public String getStateId() {return stateid;}
 public Set getSetOfBranches() {return setofbranches;}
}

class AStructure {
 private String automataid;
 private Set setofstates;

 public AStructure(String id,Set states) {
  automataid=id;
  setofstates=states;
 }
 public String getAutomataId() {return automataid;}
 public Set getSetOfStates() {return setofstates;}
}

class Automata {
 protected AStructure structure;
 protected String stateId;
```

```
 protected State initial;

 public Automata(AStructure str,String id,State s) {
  structure = str;
  stateId=id;
  initial=s;
 }
 public void processMsg(String msg) {
  State currentS = initial;
  Iterator iterA =
structure.getSetOfStates().iterator(); while(iterA.hasNext()) {
   State eachS = (State)iterA.next();
   if(eachS.getStateId().equals(stateId)) {
    currentS=eachS;
    break;
   }
  }
  Iterator iterS =
currentS.getSetOfBranches().iterator(); while(iterS.hasNext()) {
   Branch eachB = (Branch)iterS.next();
   if(eachB.getMsgCode().equals(msg)) {
    Task t=eachB.getTask();
    t.processMsg();
    stateId=eachB.getNextStateId();
    break;
   }
  }
 }
}
```

The class *Task* models the task that is performed during the transition from the current state to the next state. The task identification is stored in the class field *id*. The user of the class *Task* specifies the particular task identification as the parameter of the class constructor. The default message processing function, named *processMsg()*, just prints the task identification to the standard output file.

The class *Branch* models the arc of the state transition graph. The attributes of the state transition are the message code that triggers the state transition, the task that is performed during the state transition, and the identification of the next stable state. The corresponding fields are named *msgcode*, *task*, and *nextstateid*, respectively. These fields are set by the class constructor. The current content of these fields is returned by the functions *getMsgCode()*, *getTask()*, and *getNextStateId()*, respectively.

The class *State* models a single FSM state. The state attributes are the state identification and the set of the outgoing state transitions (the target state is irrelevant; it can be this state or some other state). The corresponding class fields are named *id* and *branches*, respectively. Their content is set by the class constructor and returned by the functions *getStateId()* and *getSetOfBranches()*, respectively.

The class *AStructure* models the FSM structure. Its attributes are the automata identification and the corresponding set of states. The corresponding class fields are *automataid* and *setofstates*. The class constructor gets particular values for these fields through its parameters. The functions *getAutomataId()* and *getSetOfStates()* return the current values of these fields.

Finally, the class *Automata* models the complete FSM. Its attributes are the FSM structure (essentially the set of sets of state transitions), the current state identification, and the initial state identification. The corresponding class fields are named *structure*, *stateId*, and *initial*, respectively. These fields are set by the class constructor.

The function *processMsg(String msg)* is the event interpreter. The input argument *msg* is the message, which triggered the state transition. The interpretation starts with the iteration through the set of states to locate the object that corresponds to the FSM current state (its identification is stored in the field *stateId*). This is a typical object-oriented approach, which avoids the unpopular *switch–case* and similar selection statements. Principally, this first iteration is really not needed and can be easily eliminated by saving the current state object instead of the current state identification. However, the first iteration is intentionally kept to make the example more informative by showing how we can use two subsequent iterations to search through the set of sets of state transitions.

The second iteration searches through the set of state transitions that correspond to the current state to locate the state transition that corresponds to the input message *msg*. After locating the state transition, it gets the object that corresponds to the state transition task and calls its *processMsg()* functions, which, in turn, prints the task identification to the standard output file.

From the program code given above, the classes *Task*, *Branch*, *AStructure*, and *Automata* are obviously generic and can be used for the construction of any FSM. Besides that, this solution enables the design and implementation of dynamically reconfigurable FSMs, because sets in Java can be dynamically updated with the corresponding task object dynamically loaded and unloaded.

We illustrate the applicability of this set of classes with the following implementation of the counter by modulo 2 in Java (the corresponding overall class architecture is shown in Figure 4.9):

```
class Task0 extends Task {
 public Task0(int ident) {super(ident);}
 public void processMsg() {System.out.println("0");}
}

class Task1 extends Task {
 public Task1(int ident) {super(ident);}
 public void processMsg() {System.out.println("1");}
}

class Task2 extends Task {
 public Task2(int ident) {super(ident);}
 public void processMsg() {System.out.println("2");}
}

class Automata2 {
 public static void main(String[]args) throws IOException {
  Automata a2 = makeAutomata();
  char ch;
  String msg;
  System.out.println("This is the example of counter by modulo 2.");
```

```
  System.out.println("The automata evolution has started...");
  while(true) {
   System.out.print("Enter input signal (0/1 and <ENTER>): ");
   ch = (char)System.in.read();
   System.in.skip(2);
   if(((ch!='0') && (ch!='1'))) break;
   if(ch=='0') msg="0"; else msg="1";
   a2.processMsg(msg);
  }
 }
 private static Automata makeAutomata() {
  Branch b11 = new Branch("0",new Task0(0),"0");
  Branch b12 = new Branch("1",new Task1(1),"1");
  Set s1 = new HashSet();
  s1.add(b11); s1.add(b12);
  state S1 = new State("0",s1);

  Branch b22 = new Branch("0",new Task1(1),"1");
  Branch b23 = new Branch("1",new Task2(2),"2");
  Set s2 = new HashSet();
  s2.add(b22); s2.add(b23);
  State S2 = new State("1",s2);

  Branch b33 = new Branch("0",new Task2(2),"2");
  Branch b31 = new Branch("1",new Task0(0),"0");
  Set s3 = new HashSet();
  s3.add(b33); s3.add(b31);
  State S3 = new State("2",s3);

  Set a = new HashSet();
  a.add(S1); a.add(S2); a.add(S3);
  AStructure as = new AStructure("0",a);

  Automata au = new Automata(as,"0",S1);
  return au;
 }
}
```

At the beginning of this example, we define the application-specific tasks, namely, *Task0*, *Task1*, and *Task2*, which are responsible for printing the counter by modulo 2 outputs (0, 1, and 2, respectively). Note that the number of tasks (three) is smaller than the number of state transitions (six) in this particular example. The application-specific *processMsg()* functions are defined by overriding the default functions.

The definitions of the classes *Task0*, *Task1*, and *Task2* are followed by the definition of the class *Automata2*, which comprises two public functions: *main()* and *makeAutomata()*. The function *main()* starts by calling the function *makeAutomata()*, which, in turn, returns the counter by the modulo 2 object, named *a2*. After that, it falls into an infinite *while* loop in which it reads the standard input file. If the input character is neither "0" nor "1," it breaks the loop and the program terminates. Otherwise, it converts an input character into the corresponding string ("0" and "1," respectively) and passes it as an input event to the event interpreter.

The function *makeAutomata()* constructs individual state transitions (instances of the class *Branch*), individual states (instances of the class *State*),

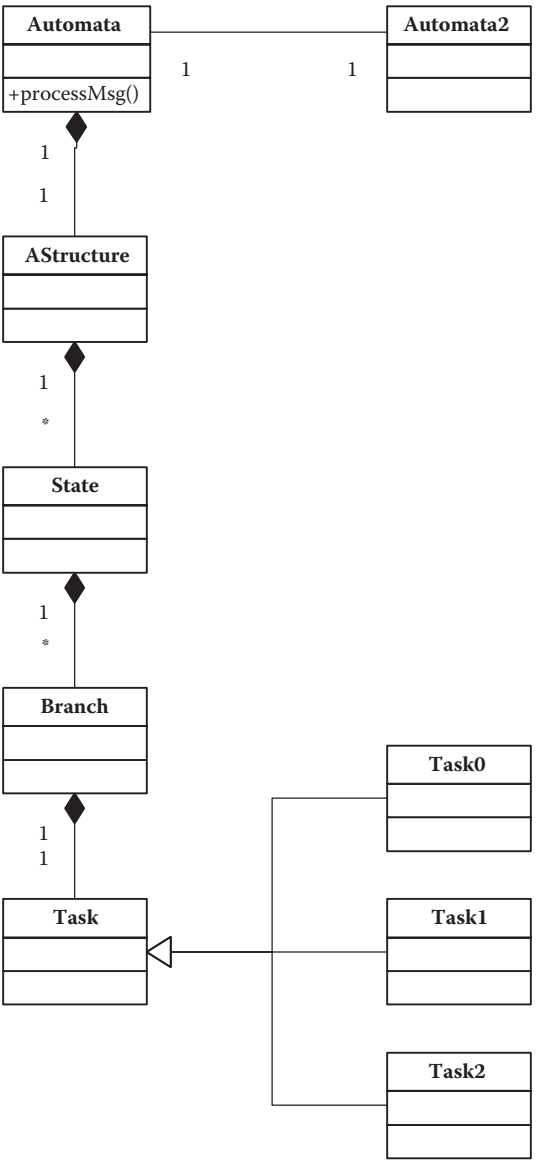


FIGURE 4.9
Static structure used in the second approach to the FSM implementation.

the counter by modulo 2 structure (an instance of the class *AStructure*), and the counter by modulo 2 itself (an instance of the class *Automata*). It first constructs the state transition *b11*, which for the input "0" moves the FSM from the state *S1* to the same state, and during that transition it performs the task *Task0*. Similarly, it constructs the state transition *b12*, which for the

input "1" moves the FSM from the state *S1* to the state *S2*, and during that transition it performs the task *Task1*. Next, it constructs the set of state transitions *s1* and the state *S1*.

Likewise, this function constructs the state transitions *b22* and *b23* and the state *S2*, as well as the state transitions *b33* and *b31* and the state *S3*. Finally, it constructs the structure of the counter by modulo 2, named *as*, and the counter by modulo 2, named *au*.

The third approach to FSM implementation, from the broad spectrum of implementations, is illustrated next. In this approach, we define the FSM structure with the corresponding class hierarchy and the set of lookup tables that map FSM inputs into the corresponding state transitions. This approach also uses message interpretation and is therefore universal, like the previous one, but it yields much better performance that is comparable with the performance of the first approach (nested **switch–case** statements).

The first idea behind this concept is to model each FSM stable state with the class that is derived from the basic class *State*. The second idea is to consider a state transition (represented with the corresponding arc of the state transition graph) as a transient (i.e., unstable) state. Each state transition is modeled with a class that is derived from the class that represents its originating stable state.

These two ideas lead to a class hierarchy with two hierarchical levels. The root of the class hierarchy is the basic class *State*. The first level of hierarchy defines the FSM stable states, whereas the second level of hierarchy defines its unstable states, i.e., state transitions.

We illustrate this approach with the example of counter by modulo 2. The corresponding class hierarchy is shown in Figure 4.10. The first hierarchy level defines the FSM stable states *S1*, *S2*, and *S3*. All of these are derived from the basic class *State*. The second level defines FSM state transitions *B11*, *B12*, *B22*, *B23*, *B33*, and *B31*. Notice that *B11* and *B12* are derived from their originating state *S1*. Similarly, *B22* and *B23* are derived from *S2*, and *B33* and *B31* are derived from *S3*.

The third idea behind this approach is that FSM evolution takes place by traversing the class hierarchy tree and by using polymorphism, one of the most powerful abstractions of object-oriented programming. Concretely, the event interpreter performs the following steps:

- Use the FSM input message (signal) and the lookup table (map), which are associated with the FSM current state, to determine the corresponding unstable state (state transition).
- Perform the application-specific task by calling the message processing function defined within the class that models the corresponding unstable state.
- Move the FSM into its next stable state.

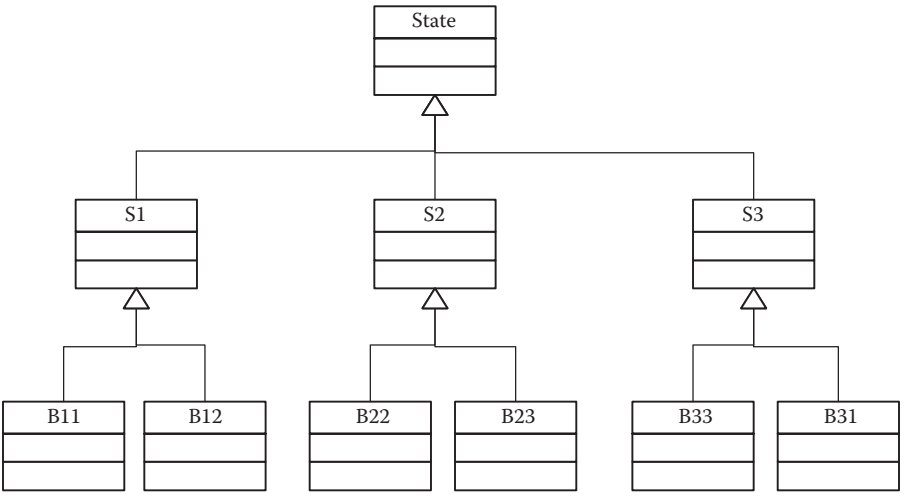


FIGURE 4.10
Counter by modulo 2 state class hierarchy.

The class hierarchy for the counter by modulo 2 is defined with the following Java module:

```
package automata;
import java.util.*;

class State {
 public State msgToBranch(String msg) {return new State();}
 public State processMsg() {return new State();}
}

class S1 extends State {
 public State msgToBranch(String msg) {
  return Structure3.getBranch("0",msg);
 }
}
class S2 extends State {
 public State msgToBranch(String msg) {
  return Structure3.getBranch("1",msg);
 }
}
class S3 extends State {
 public State msgToBranch(String msg) {
  return Structure3.getBranch("2",msg);
 }
}

class B11 extends S1 {
 public State processMsg() {
  System.out.println("Output: 0");
  return new S1();
 }
}
class B12 extends S1 {
```

```
 public State processMsg() {
  System.out.println("Output: 1");
  return new S2();
 }
}

class B22 extends S2 {
 public State processMsg() {
  System.out.println("Output: 1");
  return new S2();
 }
}
class B23 extends S2 {
 public State processMsg() {
            System.out.println("Output: 2");
            return new S3();
 }
}

class B33 extends S3 {
 public State processMsg() {
  System.out.println("Output: 2");
  return new S3();
 }
}
class B31 extends S3 {
 public State processMsg() {
  System.out.println("Output: 0");
  return new S1();
 }
}

public class Automata3 {
 private State state;

 public Automata3() {
  state = new S1();
 }
 public void processMsg (char chmsg) {
 String msg;
 if(chmsg=='0') msg="0"; else msg="1";
 state = state.msgToBranch(msg);
  state = state.processMsg();
 }
}
```

The basic class *State* has two default functions, *msgToBranch()* and *processMsg()*. Both functions return an instance of the class *State*. The fact that the instance of the class derived from the class *State* is also considered to be the instance of the class *State* that enables the event interpreter to employ polymorphism. We will return to this point shortly.

The function *msgToBranch()* is responsible for mapping the FSM input message into the corresponding state transition object. The input message in this simple example is a one-character string ("0" or "1"). The function can return any instance of the basic class *State*, but normally in this example, it should return the instance of the class *B11*, *B12*, *B22*, *B23*, *B33*, or *B31*.

The function *processMsg()* carries out the application-specific task for the given input message. It returns the FSM's next stable state. The idea is that the FSM dynamically changes its behavior. The FSM is in a certain state,

either stable or unstable, at any point in time, but it is always represented by a single object. That object is actually returned by one of these two functions, which are called in the course of FSM evolution.

Next, we define the classes that model the FSM stable states, namely, *S1*, *S2*, and *S3*. Each of these classes extends the basic class *State* and overrides the default function *msgToBranch()* with the application-specific one. These particular functions actually delegate their responsibility to the function *getBranch()* of the class *Structure3* by passing their identification ("0," "1," and "2" for *S1*, *S2*, and *S3*, respectively) and the input message to it. More precisely, these simple functions just return the unstable state object that is provided by the function *getBranch()* to their caller, and that is the event interpreter.

The stable state classes are followed by the classes that model the FSM unstable states, namely, *B11*, *B12*, *B22*, *B23*, *B33*, and *B31*. Each of these classes extends the corresponding stable state class and overrides the default function *processMsg()*, which each individual class inherits from the basic class *State*, with the application-specific one. These particular functions perform the application-specific tasks and return the corresponding next stable state object (*S1* for *B11* and *B31*, *S2* for *B12* and *B22*, and *S3* for *B23* and *B33*). The application-specific tasks in this simple example are implemented as the corresponding print statements to the standard output file.

The FSM is modeled with the class *Automata3*. This class has a single attribute named *state*, which is set by the class constructor to the FSM initial stable state, namely *S1*. Later, during the FSM evolution, it changes and can become any FSM state, either stable or unstable.

The class *Automata3* has a single function, named *processMsg()*, that is the FSM event interpreter. This function performs one state transition in two steps. In the first step, it calls the function *msgToBranch()* of the FSM current stable state object. This effectively starts the state transition by moving the FSM from its current stable state to the unstable state that corresponds to the input message. In the second step, the event interpreter calls the function *processMsg()* of the FSM unstable state, which performs the application-specific task and returns the next FSM stable state object. This effectively completes the state transition. Interestingly, the state class hierarchy in this approach is completely application-specific, whereas the event interpreter is very simple and generic and therefore can be reused in the implementations of other FSMs.

The following utility classes support the mapping of input messages to the corresponding state transitions (unstable state objects):

```
package automata;
import java.util.*;

class MapContainer {
 private String identification;
 private Map map;

 public MapContainer(String id,Map m){
  identification = id;
  map = m;
```

```
 }
 public String getId() {return identification;}
 public Map getMap() {return map;}
}

public class Structure3 {
 private static Set maps;

 public void setMaps(Set m) {
  maps = m;
 }
 public static State getBranch(String id,String msg) {
  Map m = new HashMap();
  Iterator iter = maps.iterator();
  while(iter.hasNext()) {
   MapContainer each = (MapContainer)iter.next();
   if(each.getId().equals(id)) {
    m = each.getMap();
    break;
   }
  }
  return (State)m.get(msg);
 }
}
```

The class *MapContainer* stores the map identification and the map itself in the attributes *identification* and *map*, respectively. These attributes are set by the class constructor. Their current content is available through the corresponding *get* functions.

The class *Structure3* contains a set of maps for all FSM stable states. This set is established by the function *setMaps()* and is searched by the function *getBranch()*. The input parameters of the function *getBranch()* are the map (i.e., stable state) identification and the input message. The function *getBranch()* iterates through the set of map containers, locates the one with the given identification, uses the located map to get the state transition that corresponds to the input message, and returns it to its caller.

An important feature of this approach is that it is based on Java sets and maps, which makes it an ideal environment for making dynamically reconfigurable FSMs as Java sets and maps can be dynamically updated. For example, if we want to add a new state transition *B21*, it would be sufficient to write, compile, and dynamically load a new class *B21* that represents it, and add the corresponding entry in the map that is associated to the FSM stable state *S2*.

Because the current Java version does not support a map of maps, the solution for mapping input events to the corresponding state transitions presented here is based on the usage of a set of maps. It is worth mentioning that an environment with a map of maps would enable top performance implementations based on two connected mappings. The key for the first mapping would be the FSM current stable state, whereas the key for the second mapping would be the input message. The performance of such implementations would be even better than the performance of the implementations based on nested **switch–case** statements.

The class *Environment3* uses the previously defined classes and demonstrates their usability. The corresponding Java code is the following (the overall class architecture is shown in Figure 4.11):

```
package automata;
import java.util.*;
import java.io.*;

public class Environment3 {
 public static void main(String[] args) throws IOException {
  char ch = '0';
  Automata3 a3 = new Automata3();

  Map m1 = new HashMap();
  m1.put("0",new B11()); m1.put("1",new B12());
  MapContainer M1 = new MapContainer("0",m1);

  Map m2 = new HashMap();
  m2.put("0",new B22()); m2.put("1",new B23());
  MapContainer M2 = new MapContainer("1",m2);

  Map m3 = new HashMap();
  m3.put("0",new B33()); m3.put("1",new B31());
  MapContainer M3 = new MapContainer("2",m3);

  Set maps = new HashSet();
  maps.add(M1); maps.add(M2); maps.add(M3);

  Structure3 st3 = new Structure3();
  st3.setMaps(maps);

  System.out.println("This is the example of counter by modulo 2.");
  System.out.println("The automata evolution has started...");
  while(true) {
   System.out.print("Enter input signal (0/1 and <ENTER>): ");
   ch = (char)System.in.read();
   System.in.skip(2);
   if(((ch!='0') && (ch!='1'))) break;
   a3.processMsg(ch);
  }
 }
}
```

The function *main* starts by creating the object *a3*, an instance of the counter by modulo 2. It then creates all the necessary maps and map containers, the set of maps named *maps*, the object *st3*, and an instance of the class *Structure3*. After this, it sets the set of maps by calling the function *setMaps()* and falls into an infinite *while* loop in which it reads FSM input messages and calls the event (message) interpreter until the user enters a signal that is neither "0" nor "1."

The keys for searching Java maps in this simple example are just simple strings ("0" and "1"). This Java map is a rather powerful abstraction because its key may be any class whose instances are comparable. This makes it possible to model real communication protocol messages with such classes and to build Java maps for them. Once we model the messages by the corresponding objects, FSM objects can interact with them in an object-oriented fashion.

If we want to provide a full object-oriented treatment of communication protocol messages, we must provide the corresponding serialization functions. Two

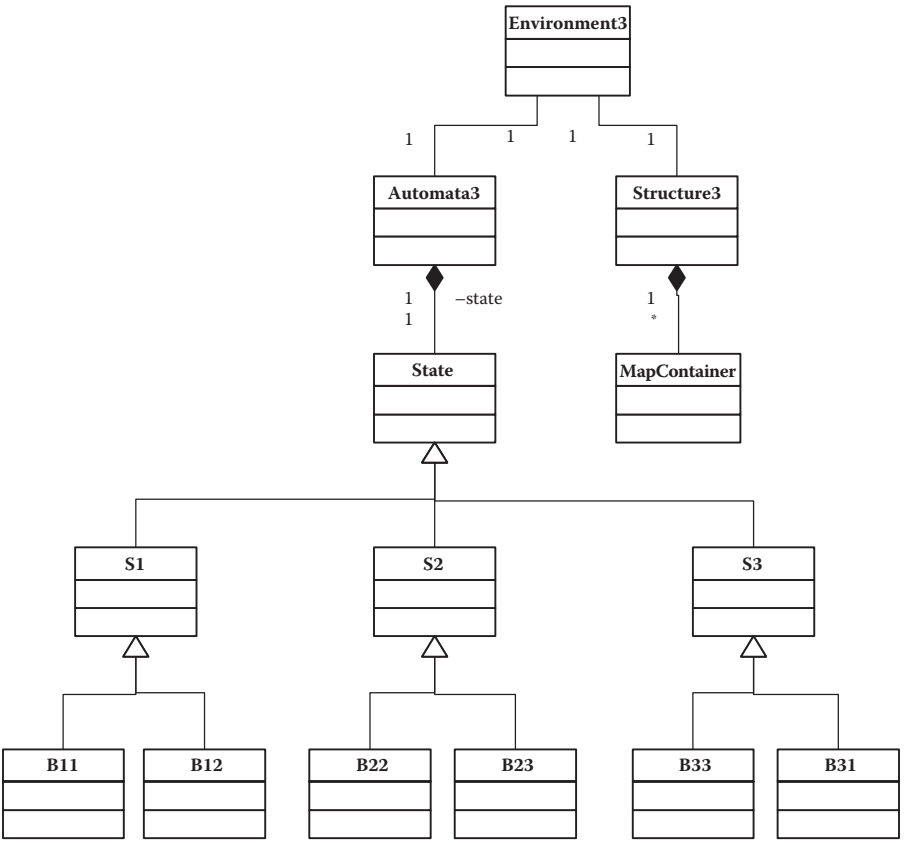


FIGURE 4.11
The static structure used in the third approach to the FSM implementation.

types of these functions are actually used. The first type is used for converting an object into a series of octets that can be transported over the communication line. The second type performs the reverse operation by converting the received series of octets into the corresponding object. If we do not provide these serialization functions, we are forced to operate directly on numbers and use **switch–case** and similar statements unpopular in the object-oriented world.

4.3 State Design Pattern

The State design pattern is one of the approaches to FSM implementation. As previously mentioned, the State pattern is shown in a separate section because it was catalogued by Gamma et al., and therefore it is not just

another example, but a well-defined and proven concept. The reader may find the complete description of the State pattern in the original book on design patterns (Gamma et al., 1995). Here we present just a brief overview and an example that demonstrates the State pattern applicability.

The original motivation to introduce this design pattern was to support objects that change their behavior as their state changes, exactly what the FSMs do. For example, when the counter by modulo 2 (Figure 4.6) is in its initial state *S1*, it produces the output 0 for the input 0, but when its state changes to *S2* or *S3*, it produces different outputs for the same input (1 in the state *S2*, and 2 in the state *S3*). Similarly, the input 1 yields the output 1 in the state *S1*, the output 2 in the state *S2*, and the output 0 in the state *S3*.

The key idea of this design pattern is to separate the FSM appearance from its behavior. We define the FSM appearance with the FSM wrapper class, which is referred to as a context. The **context** defines the user interface (a set of operations accessible by the FSM users) and contains the current FSM state object, which is one of the concrete FSM state objects.

The FSM behavior is defined with the wrapped state hierarchy. The root of this hierarchy is the generic state class, which actually defines an interface for the concrete states of the context. Each concrete state class is derived from the generic state class, and it provides the state-specific behavior of the context (FSM).

The State pattern revolves around polymorphism. Essentially, context (FSM) delegates the state-specific requests to the current state object. More precisely, each operation defined within the user interface simply calls the corresponding operation on the current state object (these operations usually have the same name). The context can pass itself as a parameter to the called operation and thus make itself accessible to the concrete state, if needed.

Typically, clients initially configure the context with state objects. Later, during the normal system operation, clients do not deal with state objects directly. Notice that either the context class or the concrete state subclass can change the context current state. Therefore, the FSM transition logic can be centralized, distributed, or hybrid.

According to the authors, the State pattern consequences are the following:

- It localizes state-specific behavior.
- It makes state transitions explicit.
- State objects can be shared.

At the end of this short overview of the State pattern, we illustrate its applicability with the simple example of a State pattern-based implementation of the counter by modulo 2. The corresponding class diagram is shown in Figure 4.12. The context in this example is the class *Automata4*. The attribute *state* holds the current FSM state object. The key function *processMsg()* delegates message processing to the current FSM state object by calling its function *processMsg()*.

The generic state class *State* defines a simple interface, which comprises a single function, *processMsg()*. Generally, such a function would define the

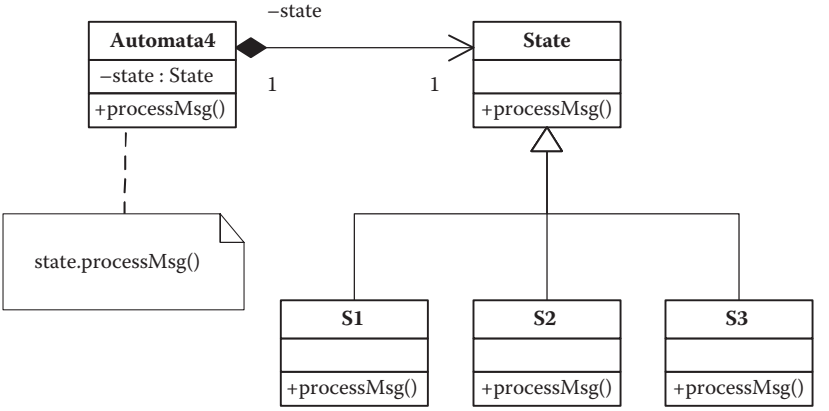


FIGURE 4.12
Static structure used by the State design pattern.

default FSM behavior, which can then be overridden in the concrete substate classes. In this simple example, as we will shortly see, no such behavior is allowed, and therefore the corresponding operation is simply empty.

The concrete substate classes *S1*, *S2*, and *S3* are derived from the generic state class *State*. Each of these classes provides a state-specific behavior by overriding the function *processMsg()* with its own particular definition. The corresponding code in Java is the following:

```
package automata4;
import java.util.*;

public class Automata4 {
 private State state;
 public Automata4() {state = new S1();}
 public void setState(State s) {state = s;}
 public void processMsg(char msg) {
  state.processMsg(this,msg);
 }
}

class State {
 public void processMsg(Automata4 a,char ch) {
 }
}

class S1 extends State {
 public void processMsg(Automata4 a,char ch) {
  if(ch=='0') {
   System.out.println("Output 0");
   a.setState(new S1());
  } else {
   System.out.println("Output 1");
   a.setState(new S2());
  }
 }
}

class S2 extends State {
```

```
 public void processMsg(Automata4 a,char ch) {
  if(ch=='0') {
   System.out.println("Output 1");
   a.setState(new S2());
  } else {
   System.out.println("Output 2");
   a.setState(new S3());
  }
 }
}

class S3 extends State {
 public void processMsg(Automata4 a,char ch) {
  if(ch=='0') {
   System.out.println("Output 2");
   a.setState(new S3());
  } else {
   System.out.println("Output 0");
   a.setState(new S1());
  }
 }
}
```

The definition of the class *Automat4* begins with the definition of the field *state*, which is used to store the FSM current state object. The class constructor sets this field to the FSM initial state object, which is an instance of the class *S1*. The function *setState()* is used by the FSM concrete state objects to change the FSM state (an example of distributed transit logic). The function *processMsg()* simply calls the corresponding function on the FSM current state object.

The class *State* defines a simple state interface with just one function—*processMsg( )*—which is empty because this example has no default behavior. The class *S1* is an example of a concrete substate class. It defines the *S1*-specific FSM behavior by overriding the function *processMsg()* that it inherits from the base class *State*. This function checks whether the input signal is 0 or 1, prints the corresponding output signal, and changes the FSM state by calling the function *setState()*. We made the context accessible by passing it as a parameter to the function *processMsg()*.

The following Java code creates the working environment for this example (given without the comments because a similar code is already explained in a previous section):

```
package automata4;
import java.util.*;
import java.io.*;

public class Environment4 {
 public static void main(String[] args) throws IOException {
  char ch = '0';
  Automata4 a4 = new Automata4();
  System.out.println("This is the example of counter by modulo 2.");
  System.out.println("The automata evolution has started...");
  while(true) {
   System.out.print("Enter input signal (0/1 and <ENTER>): ");
   ch = (char)System.in.read();
   System.in.skip(2);
```

```
    if(((ch!='0') && (ch!='1'4))) break;
    a4.processMsg(ch);
   }
  }
}
```

4.4 Implementation Based on the FSM Library

In the previous two sections, we have explored various approaches to the FSM implementations by means of simple examples. The reader should be much more familiar with FSM implementation by now, but for serious communication protocol engineering we need much more. We need a well-established working environment that will enable productive and repeatable development processes that yield maintainable products (communication protocols) of high quality.

The main measure (metrics) of quality in the context of communication protocols is their reliability, which is considered to be proportional to the number of remaining software bugs. Another important quality measure is the product performance measure with its throughput (the number of messages processed in the given interval of time) and hardware resources needed to achieve that throughput (RAM and ROM size and processor speed measured in MIPS or MHz). Generally, one of the key factors to successful software quality assurance is the quality of the software tools used in the development process. Communication protocol engineering is by no means an exception in this respect.

In this section, we present an example of the state-of-the-art working environment for the productive development of communication protocols. The environment is effectively created by an integrated development environment, which includes a C++ compiler and the domain-specific C++ library, named FSM Library. As already mentioned, the FSM Library includes two fundamental classes, *FSMSystem* and *FiniteStateMachine*. The former creates the execution platform for a group of FSMs whereas the latter is the base class for implementing individual FSMs.

The FSM Library API comprises two interfaces, which are defined by the class *FSMSystem* and *FiniteStateMachine*. The complete FSM Library programmer reference manual is given in Chapter 6. The reference manual also includes two representative implementation examples. In this section, we focus on the FSM Library concepts and internals.

The key concept behind the FSM Library is to enable productive implementations of FSMs in a uniform way. The main task of the FSM Library user is to implement the FSM state transition functions. The user does this by translating the design artifacts (statechart diagram, activity diagram, or SDL

diagram) into the corresponding C++ class function members. This translation can be done manually or with a software tool (typically used if the product performance is not critical).

The process of translation is both productive and uniform because the FSM Library provides all the functions needed to effectively construct an FSM state transition. These functions can be classified into the following function groups:

- Message handling functions (both message header and message payload handling functions). These functions support both message coding and decoding (i.e., message synthesis and analysis).
- Message sending functions.
- Timer handling functions (essentially start, stop, and restart timer).

The reader may be puzzled by the fact that the list given above does not include any message receiving functions. The FSM Library is specific in this respect. The developer does not need to explicitly call a function that receives a message (signal). Rather, the FSM execution platform (provided by the class *FSMSystem*) routes all sent messages toward their destination automata, locates the state transition function that corresponds to the message type (determined by the content of the corresponding message header field), and calls it as its subroutine. We will see shortly that the function that performs the message routing and processing (named *Start*) is actually the event interpreter.

Therefore, the FSM Library completely supports the message handling style present in the design artifacts (statecharts, activity diagrams, and SDL diagrams), which just name the input event (message) without taking care of how that event is effectively recognized (received). The FSM Library provides the class *FSMSystem* to support the straightforward implementations of design artifacts. Once provided with the class *FSMSystem*, the developers do not care how the message is received; they simply write the C++ function that performs the state transition when the message is received.

Other FSM Library specifics are the following:

- The FSM implementation is independent from the underlying real-time kernel.
- The FSM Library provides the mechanism to send messages to the dynamically allocated automata instances, which are referred to as unknown automata instances.
- The FSM Library provides public mailboxes, which can be used as message queues with different priorities.
- The FSM Library separates the message handling functions from the real-time kernel. This feature is referred to as the encapsulation of the message handling functions.

- The FSM Library treats timers as special messages, which are distinguished from the communication protocol messages by the code that determines the message type.
- The logging system provided by the FSM Library is based on the test version of the real-time kernel, which is derived from the target (final) real-time kernel.
- The FSM implementation is independent from the concrete formats of the communication protocol messages.
- The FSM Library provides automatic message buffer reallocation in cases where current buffer capacity becomes insufficient for storing additional message parameters.

The following paragraphs provide short comments on each of these FSM Library specifics. We proceed through the list of specifics from its beginning toward its end.

An important design decision was to make the FSM Library independent from the underlying run-time kernel. This decision is important because it enables easy porting of the FSM implementations to various target platforms (bare machine, UNIX, Windows NT). The internal class *KernelAPI* facilitates this independence. It represents a clean interface between the FSM implementation and the run-time system. The kernel developer must derive a new class from the class *KernelAPI* and write its real member functions by taking into account the details of the particular target platform. An example of such implementation is shown later in this section.

The second FSM Library-specific feature is related to the beginning of the communication between two FSMs, namely, FSM A and FSM B, where the former has the active role and the latter is passive. The problem is simple if A always communicates with the same B, but it becomes more complex if B is not known in advance (B is an unknown FSM). Consider a pool of FSMs, where each is capable of performing the same task. FSM A is principally interested in engaging with any instance from the pool that is free.

The FSM Library facilitates the communication with the unknown automata by placing all relevant data into the header of the message that is sent to it. The message destination is set to the special code, named *UNKNOWN_AUTOMATA*. The function member *Start* of the class *FSMSystem* recognizes this code and dynamically allocates an automata instance, which will be the message destination and therefore involved in the further communication with the message originator. In the case when there are no free automata instances available in the pool, the function *Start* calls the special function *NoFreeInstances*, which is responsible for the recovery procedure. Typically, this function informs the message originator about the automata instance outage by sending it an appropriate signal, such as *NAK*, *DISCONNECT*, and so on.

The third FSM Library-specific feature is the provision of general purpose mailboxes, which can be used both as public mailboxes and private mailboxes. The former are actually FIFO message queues that contain messages for various destinations, whereas the latter contain messages for a single destination, which is an FSM that owns the private mailbox. Generally, we can use only a single public mailbox to enable the communication between all FSMs present in the system. Such a solution can suffice in the case of simple systems with a small number of FSMs and soft real-time requirements. However, a single public mailbox may not be sufficient in the case of more complex systems because the FSM Library mailbox is just a FIFO message queue without any support for message prioritization.

The absence of message prioritization can lead to a case where an FSM processes an outdated message instead of processing the corresponding timeout message, just because the outdated message is ahead of the timeout message in the public mailbox. Such cases can lead to dysfunctional behaviors that are not caused by design oversights but, rather, inappropriate implementation.

The regular method of supporting message prioritization in the FSM Library-based implementations is to use more public mailboxes that are assigned different priorities. For example, we can use three public mailboxes for three different priorities. These three public mailboxes are effectively treated as three FIFO message queues with different priorities (e.g., high, medium, and low). We can select a strategy of using private mailboxes instead. We can also mix public and private mailboxes if we wish. Actually, the function *Start* (the member of the class *FSMSystem*) treats them equally. In its loop, it searches all the mailboxes for messages. The effective mailbox priority is determined by the order of that search (i.e., it starts from the mailbox index 0).

The fourth FSM Library–specific feature is the encapsulation of the message handling functions. Generally, real-time kernels can store the message source and destination information in the message header or in the separate data structure. By separating the message handling functions into a group that handles the message header and a group that handles the message payload, the FSM Library provides complete FSM implementation independence from the message source and destination information location.

An additional enhancement related to the message destination provided by the FSM Library is the support for sending messages to the *left* or to the *right* FSM. The abstraction of the *left* and *right* FSM originally comes from SDL. If the SDL symbol for sending a message points to the left, we say that the message is sent to the *left* FSM. Similarly, if the symbol points to the right, we say that the message is sent to the *right* FSM.

The internal class *KernelAPI* provides the functions *SendMessageLeft* and *SendMessageRight*, which are inherited by the class *FiniteStateMachine*, to support this abstraction. These two functions enable the direct coding of corresponding parts of SDL diagrams, and the resulting C++ code has a great similarity with the original SDL diagrams. For example, consider the following snippet of C++ code that corresponds to a state transition:

```
StopTimer(FE4_TIMER1);
DisconnectRingTone();
PrepareNewMessage(0x00,r2_SetupRespConf);
SendMessageLeft();
StartChargingIncoming();
Connect();
SetState(FE4_ACTIVE);
```

The call of the function *SendMessageLeft()* above is a direct encoding of the corresponding left-pointing SDL graphical symbol. This snippet of code is a typical state transition implementation based on the FSM Library, which is rather short and easy to read and map to the original design model. These are two key implementation features that ensure productivity and quality.

The fifth FSM Library–specific feature is that it treats timers as special messages, distinguished from the communication protocol messages by the code that determines the message type. Some of the message header parameters are meaningless for timers. The corresponding message header fields are used by the FSM Library API functions related to timers to store the data specific for individual timers, such as timer duration.

All timers used by a certain FSM type must be initialized in the FSM class function member *Initialize()* by calling the function *InitTimerBlock()* (see Section 6.8.74). The parameters of this function are the timer identification, the timer duration, and the identification of the message to be sent when the timer expires. In response to a series of *InitTimerBlock()* calls, the system creates the corresponding array of timers. The identification of a timer effectively becomes the index of this array.

Once initialized, the timer can be started by the function *StartTimer()*, stopped by the function *StopTimer()*, restarted by the function *RestartTimer()*, or checked by the function *IsTimerRunning()*. All these functions have a single parameter, the identification of the timer. Therefore, the resulting C++ code resembles the original design model to a great extent. Moreover, when the timer expires, the corresponding message is automatically sent to the FSM that started it, which processes this message in the same fashion as all other messages. This feature also contributes to the similarity of the resulting C++ code and the original design model.

The sixth FSM Library–specific feature is that the logging subsystem provided by the FSM Library is based on the test version of the real-time kernel, which is derived from the target (final) real-time kernel. The logging subsystem is important in communication protocol engineering because certain design oversights or implementation errors become evident only in complex circumstances, which can happen only after long run-time periods. Typically, such circumstances are difficult to repeat and therefore developers normally use log files to backtrack the sources of errors once they occur.

The FSM Library provides a complete logging subsystem that is used both during system testing and normal system exploitation. The internal class *LogAutomata* defines the necessary set of functions. FSM tracing is based on the interception of all relevant internal functions, such as FSM state updating,

message processing, timer management functions, and so on. Automatic logging of various events makes the resulting log file outlook uniform, and thus easy to read by any member of the development team. All logging events are prioritized, which helps developers to easily define exactly which events they want to trace.

Traditionally, log files are located on mass storage devices such as hard disks or flash memory. The FSM Library introduces an enhancement in this respect. The internal class *LogInterface* defines the interface between the system implementation and the concrete logging media, such as the conventional log file, the TCP/IP connection to the logging server, and so on. Logging to the concrete media is provided by a subclass that is derived from the base class *LogInterface*. Examples of such classes are the classes *LogFile* and *LogTCP*.

The seventh FSM Library–specific feature is that the FSM implementation is independent of the concrete formats of communication protocol messages. The feature is facilitated by the internal class *MessageHandler*, which provides a set of generic functions for manipulating message parameters. Basically, two families of these functions exist, namely, *get* and *add*. The former returns the value of the given parameter, whereas the latter adds the given message parameter to the message. The parameter is specified with its identification (code) and its value.

The class *MessageHandler* uses the class *MessageInterface*, which is an abstract class that defines the interface for the abstract message format. Normally, the developer derives a class from the class *MessageInterface* for each concrete message format and writes its function in accordance with the format-specific details. An example of such a class is the class *StandardMessage*, which models a message that comprises a sequence of octets (characters). Such an approach centralizes message handling functionality. This centralization eliminates code redundancy and increases code coverage. Additionally, development team productivity is increased because message handling functions and FSMs can be developed in parallel.

The eighth and last FSM Library-specific feature is that it provides automatic message buffer reallocation in cases where the current buffer capacity becomes insufficient for storing additional message parameters. Although this functionality is rather easily implemented, it is important because it makes the process of message creation completely transparent. The programmer just adds parameters to the new message as needed, without having to take care about the size of the free space in the corresponding buffer. This detail is completely hidden by the message handling functions.

4.4.1 Using the FSM Library

Using the FSM Library is rather easy. It helps a lot in both the design and implementation phases of the development process. The author's experience shows that both students and engineers working in the industry can start using it only after a couple of days of training. Actually, it does not take more than writing one example based on the FSM Library to start using it. Besides

that, it is a well-established working environment that has been used in a series of the real-world projects for the industry.

When it comes to design, the FSM Library greatly simplifies matters by providing two fundamental classes, *FSMSystem* and *FiniteStateMachine*. The existence of these two classes makes the system static structure well known from the start (Figure 3.5). Each protocol is modeled by the subclass derived from the base class *FiniteStateMachine*. The resulting FSM is executed by the event interpreter, which is hidden inside the class *FSMSystem*. These two classes practically encapsulate all domain-specific design patterns needed for designing a communication protocol.

The overall result is that the class diagram is almost not needed at all, at least not for realistic communication systems that comprise less than a dozen communication protocols. Even for very complex communication systems based on the FSM Library, the class diagram can be used more as an accompanying document. The most informative part of such a class diagram would be the one that specifies the mailboxes present in the system, as well as the timers used by individual FSM types.

The real valuable design artifacts for the paradigm based on the FSM Library are the complete models of the system behavior in the form of the activity, statechart, or SDL diagrams. This is the case because the FSM Library *de facto* specifies the skeleton of the system static structure, but it does not (and cannot) specify the complete system behavior. It provides only primitive behavior from which we can build more complex behavior, in particular, the state transitions.

Once we have finalized the detailed design diagrams (activity, statechart, or SDL diagrams), we are ready to proceed to the implementation phase of the development process. The main task of implementing FSMs by using the FSM Library, besides writing the initialization function and a couple of simple auxiliary functions, is the encoding of state transitions by using the set of primitives provided within the FSM Library application programming interface (see Section 6.8). A good thing about these primitives is that they provide mapping of SDL steps in almost a one-to-one manner. The names of the primitives are almost self-documenting, at least after the short experience you get by using them. The code resembles the original design artifacts (especially SDL diagrams). All these attributes help any member of the development team to read, understand, and continue the work that was done by some other member of the development team, especially if they have the design artifact at their disposal.

It is also worth mentioning that besides forward engineering, the FSM Library helps backward engineering too. This is especially true if the backward engineering is done by hand. Using software tools for that purpose is also possible if the development team strictly obeys certain coding guidelines. The key for successful forward and backward engineering is a well-defined API (see Section 6.8).

We demonstrate the usage of the FSM Library API by the examples at the end of this chapter, as well as with the examples at the end of Chapter 6.

4.4.2 FSM Library Internals

This section describes the FSM Library internals. The main FSM Library components are the following:

- The class *FSMSystem*
- The class *FiniteStateMachine*
- The real-time kernel

The class *FSMSystem* provides the following functionalities:

- Initialization of the FSM objects: The result is a set of the corresponding transition tables, which determine which state transitions are triggered by the individual events (messages).
- Routing of messages: This component locates the message destination FSM, looks up its state transition table to find the state transition that corresponds to the message type, and calls the corresponding function as its subroutine.
- Public mailbox prioritization: The public mailbox priority decreases as its identification increases. The identification is actually the index of the corresponding mailbox array. The public mailbox with the identification 0 has the highest priority.
- Allocation of FSMs from the pool of FSMs: If the message destination is an unknown object of a certain type, a free FSM from the corresponding pool is allocated to process that message.

The class *FiniteStateMachine* provides the following functionalities:

- Maintaining the current state variable (the field member of this class)
- Maintaining the state transition table
- FSM evolution support by providing the address of the state transition function that corresponds to the incoming message type
- Message handling (message checking, parsing, and creation)
- Message exchange (the message send operation is explicit whereas the message receive operation is implicit)
- Memory management (supports requesting and releasing buffers for messages)
- Timer management (supports starting, stopping, restarting, and testing timers)

The functionalities provided by the real-time kernel are inherited by the class *FiniteStateMachine* (message exchange, buffer, and timer management). The following subsections describe the internals of these three components.

4.4.2.1 FSMSystem Internals

As already mentioned, the class *FSMSystem* provides the execution platform for all FSMs present in the system. The list of concrete functionalities provided by this class is already given in the previous section. The heart of the class *FSMSystem* is the function *Start*, which actually provides all the listed functionalities. Essentially, it is the event (message) interpreter. Its program code in C++ is as follows:

```
void FSMSystem::Start(){
 SystemWorking = true;
 while(SystemWorking) {
  Sleep(1);
  for(uint8 i=0; i<NumberOfMbx; i++) {
   uint8 *msg = GetMsg(i);
   if(msg == NULL){
    continue;
   }
   uint8 automataType = GetMsgToAutomata(msg);
   if(((automataType > NumberOfAutomata) ||
     (NumberOfObjects[automataType] == 0))){
    // Error handling
    DiscardMsg(msg);
    continue;
   }
   uint32 objNum = GetMsgObjectNumberTo(msg);
   if(objNum == UNKNOWN_AUTOMATA){
    ptrFiniteStateMachine object =
     FreeAutomata[automataType].Get();
    if(object != 0) object->Process(msg);
    else
     (Automata[automataType][0])->NoFreeObjectProcedure(msg);
    continue;
   }
   else if(objNum > NumberOfObjects[automataType]) {
    // Error handling
    DiscardMsg(msg);
    continue;
   }
   else {
    (Automata[automataType][objNum])->Process(msg);
   }
  }
 }
}
```

The function *Start* initially sets its field member *SystemWorking* to the value *true* and enters the loop, which is executed while *SystemWorking* has the value *true*. Once this variable is set to the value *false* (this is exactly what the API function *StopSystem()* does), the function *Start* exits the loop and terminates. Because this function is the FSM event interpreter, once it stops, the whole system stops.

Inside the *while* loop, this function enters the nested *for* loop in which it checks all mailboxes for messages. This *for* loop starts from the mailbox with the identification (index) 0, thus making it the highest priority mailbox. As it proceeds toward the identification *NumberOfMbx*, the priority of the corresponding mailboxes decreases.

Once it finds a message in the mailbox, it exits the nested *for* loop and continues with determining the destination automata (FSM) type identification by calling the function *GetMsgToAutomata()*. If the identification is invalid (greater than the configuration parameter *NumberOfAutomata*) or if no instances of that type are found, the function discards the message by calling the function *DiscardMsg()* and continues the main loop.

If the automata type identification is valid and at least one instance of that type is found, the function *Start* determines the destination object identification by calling the function *GetMsgObjectNumberTo()*. If this identification is equal to *UNKNOWN_AUTOMATA*, the function *Start* tries to allocate an object from the pool of objects of the given type by calling the function *Get()* on the object of that type.

If at least one free object is found in the pool (actually an array of objects of the given type), the function *Get()* will return the identification (array index) of the first one and, in turn, the function *Start* will call its function *ProcessMsg()*. Behind the scenes, the function *ProcessMsg()* locates the state transition that corresponds to the message type, calls it its subroutine, and continues the main loop. If no free objects are in the pool, the function *Start* discards the message and continues the main loop.

Finally, if the message destination is a known object (its identification is not equal to *UNKNOWN_AUTOMATA*), the function *Start* checks if its identification is valid (not greater than the configuration parameter *NumberOfObjects*[*automataType*]). If the object identification is valid, the function Start calls object function *ProcessMsg()* and continues the main loop.

4.4.2.2 FiniteStateMachine Internals

The class *FiniteStateMachine* is at the top of the FSM Library class hierarchy (Figure 4.13). It hides the details of the FSM Library internal static structure from its user. The class *FiniteStateMachine* inherits logging-related functionality from the class *LogAutomata* (shown as the left branch of the class hierarchy in Figure 4.13). Alternately, the class *FiniteStateMachine* inherits the buffer, timer, and message management functionality from the class *KernelAPI* (shown as the right branch of the class hierarchy in Figure 4.13). Both *FiniteStateMachine* and *KernelAPI* inherit the message management functionality from the class *MessageHandler*.

The class *LogAutomata* conceptually uses the logging services provided through the interface created by the class *LogInterface*. The logging services are provided in run-time reality by the object that is an instance of a subclass, which is derived from the base class *LogInterface*. Figure 4.13 shows two examples of such classes, namely, *LogFile* and *LogTCP*. The former provides the recording of log events into the file located on some mass storage device. The latter uses the TCP/IP network to send log events packed into messages to the logging server, which, in turn, writes the log events to a file, perhaps located on its hard disk.

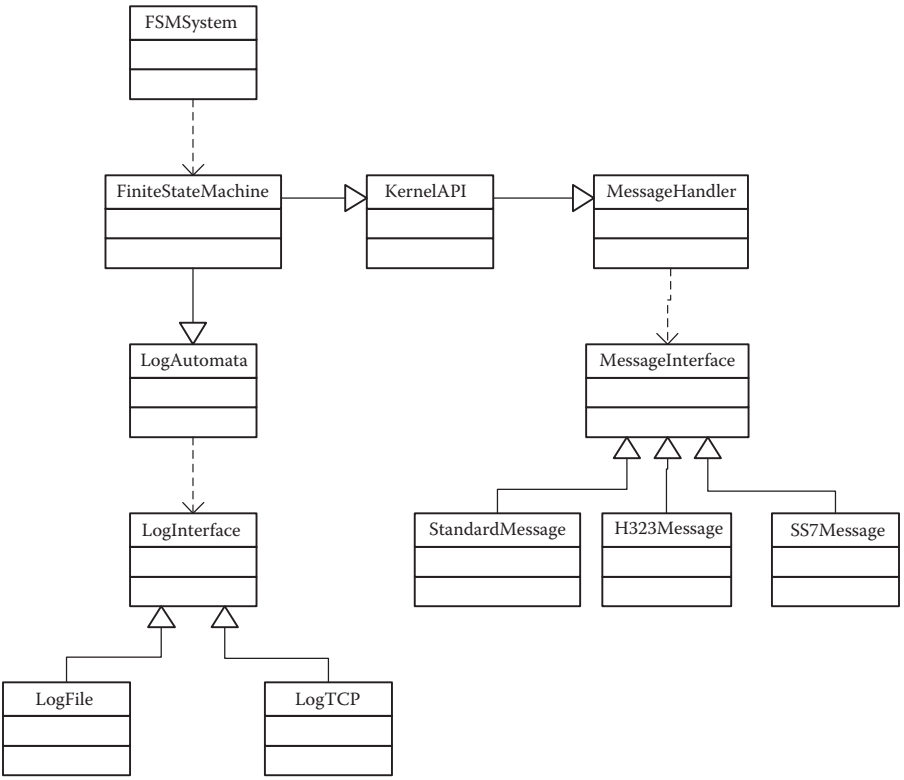


FIGURE 4.13
Internal FSM Library static structure.

Similarly, the class *MessageHandler* uses services of the abstract interface provided by the class *MessageInterface*. The real providers of the message handling services are subclasses derived from the base class *MessageInterface*. Figure 4.13 shows three examples of such classes, namely, *StandardMessage*, *H323Message*, and *SS7Message*. In the examples in this book, we use the class *StandardMessage*, which creates an abstraction of the message comprising a series of octets (characters) that can be partitioned into an arbitrary number of message fields (carrying message parameters) of arbitrary size (given as a number of octets).

In the text that follows, we cover the most important details of the class *FiniteStateMachine*, *KernelAPI*, and *MessageHandler*. The effect of this top-down approach is that we introduce first the functionality solely provided by the class *FiniteStateMachine*, then the functionality that the class *FiniteStateMachine* inherits from the class *KernelAPI*, and finally the functionality that the class *FiniteSateMachine* inherits from the class *MessageHandler*.

The class *FiniteStateMachine* comprises all attributes and operations necessary for the definition and evolution of a single FSM. The FSM state is modeled with the structure *SState*:

```
struct SState {
 SState(uint16 maxNumOfProceduresPerState);
 ~SState();
 bool StateValid; // if true, data are valid
 unsigned short NumOfBranches; // number of branches in a state
 // procedure for processing unexpected message
 PROC_FUN_PTR UnexpectedEventProcPtr;
 SBranch* PBranch; // pointer on data for each branch
};
```

The field *NumOfBranches* contains the number of outgoing state transitions (branches) for the corresponding state. The field *UnexpectedEventProcPtr* is a pointer to the C++ function that handles the reception of unexpected messages. Finally, the field *PBranch* contains a pointer to the array of the *SBranch* instances, which model individual outgoing state transitions. The structure *SBranch* definition is the following:

```
struct SBranch {
 uint16 EventCode; // message code
 PROC_FUN_PTR ProcPtr; // message processing function
};
```

The field *EventCode* contains the code of the event (message) that triggers this state transition. The field *ProcPtr* contains the pointer to the C++ function that performs the actions during this particular state transition.

Generally, an FSM can use a number of timers. Each timer is represented with an instance of the structure *TimerBlock*:

```
struct TimerBlock {
 TimerBlock(uint16 v, uint16 s) :
  Count(v), SignalId(s), Valid(false), TimerBuffer(0){}
 TimerBlock() :
  Count(INVALID_32), SignalId(INVALID_16), Valid(false),
  TimerBuffer(0) {};
 uint32 Count; // in time slices
 uint16 SignalId; // message code
 bool Valid; // if true, data is valid
 ptrBuff TimerBuffer; // Ptr to timer buffer
};
```

The field *Count* defines the timer duration, the field *SignalId* defines the code of the message (signal) that is generated when the timer expires, the field *Valid* is set if the timer is running, and the field *TimerBuffer* contains the pointer to the buffer used by the timer expiration message.

The main private field members of the class *FiniteStateMachine* are as follows:

```
class FiniteStateMachine : public KernelAPI, LogAutomate {…
 private:
```

```
uint16 NumOfStates; // Number of FSM states
uint16 NumOfTimers; // Number of timers
uint16 MaxNumOfProcPerState; // Max. no. of branches
SState *States[MAX_STATE_NO]; // State data
uint32 ConnectionId; // Current connection
uint32 CallId; // Current call
uint8 State; // Current state
```

The fields *NumOfStates*, *NumOfTimers*, and *MaxNumOfProcPerState* are the dimensions of the corresponding arrays. They define the number of FSM states, the number of timers it uses, and the maximum number of branches, respectively. The field *States* is an array of pointers to the instances of the structure *SState* that contains pointers to arrays of instances of the structure *SBranch*. This data structure corresponds to the FSM state transition table.

The field *ConnectionId* carries the domain-specific name but actually contains the FSM object identification that is unique within the scope of objects of the same type. During the system initialization, the class *FSMSystem* creates the array of FSM objects of the same type. The index of the object in that array is written into this field at that time. This identification can be used as appropriate for the application at hand. The FSM Library user can take advantage of the fact that all message sending functions automatically copy the content of this field into the object identification field of the message header.

The field *CallId* carries another domain-specific name but it can be used for various purposes in various applications. In contrast to the field *ConnectionId* whose uniqueness is limited to the scope of a single FSM type, the value of the field *CallId* is unique in the scope of the whole system. Traditionally, it has been used to identify a single call, but generally it can be used to identify any communication process of interest. Like the field *ConnectionId*, this field is also copied by the message sending functions to the message header automatically.

Finally, the field *State* is the FSM current state identification, which is the value of the index of array defined in the field *States*. This field defines the context of the FSM.

As already mentioned, the FSM Library supports the abstraction of the *left* and *right* FSM. The message sending functions, namely *SendLeftAutomata()* and *SendRightAutomata()*—originally defined in the class *KernelAPI*—require data about the *left* and *right* FSM. Relevant *FiniteStateMachine* attributes are as follows:

```
// Left automata data
uint8 LeftMbx; // left mbx id
uint8 LeftAutomata; // left automata
uint8 LeftGroup; // left group
uint32 LeftObjectId; // left object
// Right automata data
uint8 RightMbx; // right mbx id
uint8 RightAutomata; // right automata
uint8 RightGroup; // right group
uint32 RightObjectId; // right object
```

We finish the overview of the *FiniteStateMachine* internals with its initialization and control functions:

```
FiniteStateMachine(
 uint16 numOfTimers = DEFAULT_TIMER_NO,
 uint16 numOfState = DEFAULT_STATE_NO,
 uint16 maxNumOfProceduresPerState = DEFAULT_PROCEDURE_NO_PER_STATE);
virtual void Initialize(void) = 0;
void InitEventProc(uint8 state, uint16 event, PROC_FUN_PTR fun);
void InitUnexpectedEventProc(uint8 state, PROC_FUN_PTR fun);
PROC_FUN_PTR GetProcedure(uint16 event);
virtual void NoFreeInstances() = 0;
virtual void Process(uint8 *msg);
void FreeFSM();
```

The class constructor first sets the number of timers, the number of states, and the maximal number of branches per state. It then calls the function *Initialize()*, provided by the user. This function typically uses a series of calls to functions *InitEventProc()* and *InitUnexpectedEventProc()*. The former defines the state transition function for the given state and message type whereas the latter defines the unexpected message handler for the given state.

The function *GetProcedure()* is a control function that returns the address of the state transition function for the given message type in the current state. The function *NoFreeInstances()* is a recovery function that is called in cases where no more free objects of this type are found. The function *Process()* is the prototype of the state transition function. The function *FreeFSM()* releases the FSM object by returning it to the pool of objects of this type.

The class *KernelAPI* provides the following groups of functions:

- Initialization functions
- Memory management functions
- Message management functions
- Timer management functions

The initialization functions provided by the class *KernelAPI* are its constructors (see Section 6.8) and the function *setKernelObjects*, whose prototype is as follows:

```
void setKernelObjects(TPostOffice *o, TBuffers *b, CTimer *t);
```

The parameters of this function are the pointers to the objects that comprise the system mailboxes, buffers, and timers. These objects will be described in the next section.

The memory management functions provided by the class *KernelAPI* are the following:

```
uint8 *GetBuffer(uint32 length);
void RetBuffer(uint8 *buff);
```

```
bool IsBufferSmall(uint8 *buff, uint32 length);
uint32 GetBufferLength(uint8 *buff);
```

The function *GetBuffer()* returns the pointer to the buffer of the sufficient size (not less than specified by its parameter). The function *RetBuffer()* releases the given buffer. The function *IsBufferSmall()* checks the size of the given buffer. The function *GetBufferLength()* returns the size of the given buffer.

The message management functions provided by the class *KernelAPI* are the following:

```
void Discard(uint8* buff);
void SetMessageFromData();
void SendMessage(uint8 mbxId);
void SendMessage(uint8 mbxId, uint8 *msg);
void SendMessageLeft();
void SendMessageRight();
void ReturnMsg(uint8 mbxId);
```

The function *Discard()* releases the given message. The function *SetMessageFromData()* copies the data about this FSM (type, group, and instance identifications) to the corresponding fields of the new message header. According to the FSM Library terminology, the current message is the one that has been received and processed, whereas the new message is the message that is currently under construction (and will be subsequently sent).

The function *SendMessage(uint8 mbxId)* sends the new message to the given mailbox. The function *SendMessage(uint8 mbxId, unit8 *msg)* sends the given message to the given mailbox. The functions *SendMessageLeft()* and *SendMessageRight()* send the new message to the left and right automata, respectively. The function *ReturnMsg()* sends the current message to the given mailbox.

The timer management functions provided by the class *KernelAPI* are as follows:

```
uint8 *StartTimer(uint16 code, uint32 count, uint8 *info=0);
void StopTimer(uint8 *timer);
bool IsTimerRunning(uint8 *timer);
```

The function *StartTimer()* starts the given timer by setting its duration and the corresponding message buffer. The function *StopTimer()* stops the given timer. The function *IsTimerRunning()* checks if the given timer is running.

The interface defined by the class *MessageHandler* comprises the following two parts:

- Message header handling
- Message payload handling

The message header handling part provides getting and setting functions for the individual message header fields. The main message header fields are as follows:

- *MSG_FROM_AUTOMATA*: the identification of the originating FSM type
- *MSG_TO_AUTOMATA*: the identification of the destination FSM type
- *MSG_CODE*: the identification of the message type
- *MSG_OBJECT_ID_FROM*: the identification of the originating FSM object
- *MSG_OBJECT_ID_TO*: the identification of the destination FSM object
- *MSG_CALL_ID*: the identification of the application-specific communication process
- *MSG_INFO_CODING*: the identification of the message format type
- *MSG_LENGTH*: the message payload length in octets

The timer message is a special message. If the timer expires, it is sent to the same FSM that created it. Because of this, the message header fields *MSG_FROM_AUTOMATA* and *MSG_OBJECT_ID_FROM* are not needed, and thus can be used to hold information about the timer duration and the destination mailbox identification.

The class *MessageInterface* defines the set of abstract functions that handle the message payload. The key idea behind the abstraction introduced by the class *MessageInterface* is the generic message parameter definition, which is independent from the particular message format. Each message parameter is uniquely defined by the following data:

- The message parameter identification
- The message parameter length (size)
- The message parameter value (content)

Depending on the message format type, the first and the second items listed may be implicit or explicit. Some of the messages carry the message parameter identification and length, and some do not. However, all three items must be known to the message handling functions.

Another important fact related to the message format is that particular message formats can be disassembled to a series of primitive elements of the following types:

- Byte (1 byte)
- Word (2 bytes)

- DWord (4 bytes)
- Sequence of bytes (*n* bytes)

Therefore, the class *MessageInterface* includes the functions that provide access to these primitive types of information. These functions can be partitioned into the following two groups:

- Current message handing functions
- New message handling functions

The current message handling functions are as follows:

```
uint8 *GetParam(uint8 paramCode);
bool GetParamByte(uint8 paramCode, BYTE &param);
bool GetParamWord(uint8 paramCode, WORD &param);
bool GetParamDWord(uint8 paramCode, DWORD &param);
```

The first function returns a pointer to the parameter (sequence of octets) whose identification (*paramCode*) is given. The next three functions return the requested parameter of the size *Byte*, *Word*, and *DWord*, respectively. The new message handling functions are as follows:

```
uint8 *AddParam(uint8 paramCode, uint8 paramLength, uint8 *param);
uint8 *AddParamByte(uint8 paramCode, BYTE param);
uint8 *AddParamWord(uint8 paramCode, WORD param);
uint8 *AddParamDWord(uint8 paramCode, DWORD param);
bool RemoveParam(uint8 paramCode);
```

The first four functions add the given sequence of octets, *Byte*, *Word*, and *DWord* parameter, respectively, to the new message. The function *RemoveParam()* removes the parameter—whose identification is given—from the message.

Each message handling function consists of two parts, a preparation part and an operation part. The preparation part of the current message handling functions includes preparing temporary data and message parsing. In case of message syntax errors, message handling functions report an error by returning the *value false*. The preparation part of the new message handling functions includes allocation of the message buffer and initialization of the message header fields *MSG_CODE*, *MSG_INFO_CODING*, and *MSG_LENGTH* (initially set to 0).

4.4.2.3 Kernel Internals

As already mentioned, the class *FiniteStateMachine* is independent of the particular real-time kernel with the introduction of the API defined by the class *KernelAPI*. Generally, the class *FiniteStateMachine* can use services provided by any real-time kernel that is a subclass of the class *KernelAPI*. In this section, we cover the internals of one such kernel (a default one), which is simply referred to as *Kernel*.

Figure 4.14 shows the static structure of *Kernel*. The root of the structure is the class *KernelAPI*, which acts as the wrapper of *Kernel*. This class contains pointers to the following three main parts of *Kernel*:

- Memory manager
- Message manager
- Time manager

The interfaces to these three resource managers are defined by the classes *TBuffers*, *TPostOffice*, and *CTimer*, respectively. The memory manager comprises the class *TBuffers* and a set of instances of the class *TBufferQueue*. The message manager consists of the class *TPostOffice* and a set of instances of the class *TMailBox*. The time manager is implemented by the class *CTimer* itself.

The class *TBuffers* creates an abstraction of a set of buffer pools. The size of the buffers in the pool is the same, but these sizes are different between the pools.

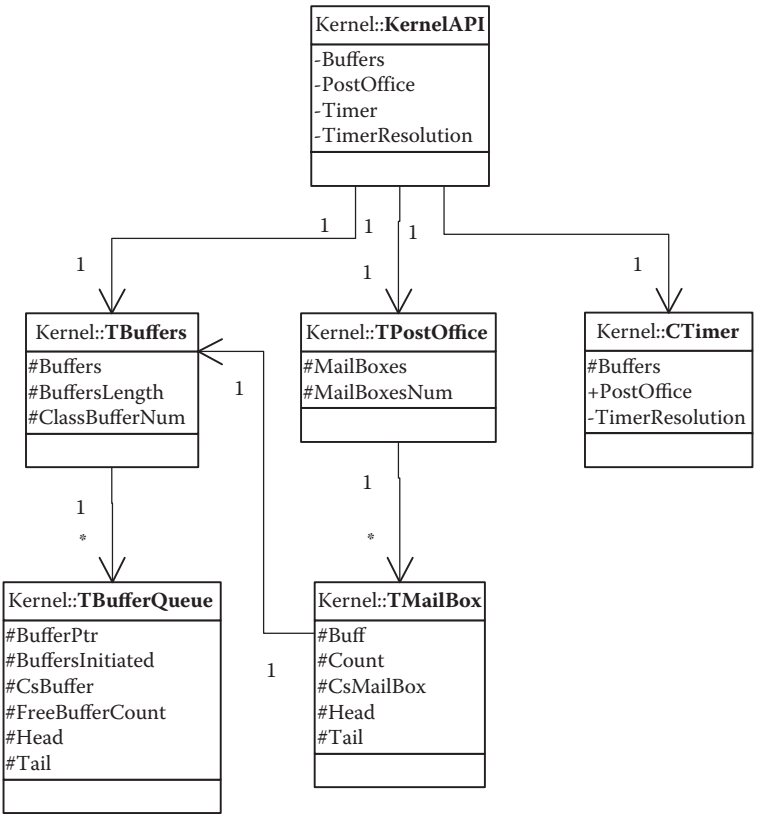


FIGURE 4.14
Internal *Kernel* static structure.

For example, we can have three pools with three different sizes, namely, small, medium, and large. The class *TBufferQueue* models one such a pool.

The constructor of the class *TBufferQueue* initially allocates an array of bytes (uint8), which is the actual memory space that accommodates the memory pool:

```
// calculate memory size for all buffers and get memory for them
memSize = bufferLength + BUFF_HEADER_LENGTH;
memSize *= buffersNo;
BufferPtr = new uint8[memSize];
```

This memory space is then partitioned into individual memory buffers that are added to the list of free buffers that actually represent the buffer pool. A buffer consists of the buffer header and the space for useful data. The buffer header comprises the pointers to the previous and to the next element in the list and the buffer code that indicates buffer size. Each buffer pool is defined with the pointer to the list of free buffers and the size of the buffers in that list. The class *TBuffers* holds the array of pointers to the instances of the class *TBufferQueue* (in the field member *Buffers*), as well as the array of the corresponding buffer sizes (in the field member *BuffersLength*).

The function *GetBuffer()* provided by the class *KernelAPI* first searches the field *BuffersLength* to find the pool of buffers of the sufficient size. It then gets the buffer from the head of the list of free buffers and returns the pointer to it. The function *RetBuffer()* uses buffer code from its header to return the buffer by adding it to the end of the corresponding list.

The class *TPostOffice* stores the array of pointers to the corresponding mailboxes. A mailbox is implemented as an instance of the class *TMailBox*. Actually, the class *TMailBox* is very similar to the class *TBufferQueue*. The main difference between them is that the former provides atomic (uninterruptible) access to the list of messages. This feature is needed because the list of messages is a resource shared by two concurrent processes, namely the event interpreter and the time interrupt routine.

The atomic mailbox access is ensured by two virtual functions, namely *MbxLock()* and *MbxUnlock()*. The former function locks the mailbox and the latter unlocks it. These functions ensure the FSM Library's portability. They can be implemented by the use of semaphores provided by the local operating systems. (The FSM Library supports OS Linux® and Windows® NT at the moment.)

The class *CTimer* is the most target-platform-dependent part of *Kernel*. It consists of two parts, a platform-dependent part and a platform-independent part. The platform-dependent part comprises the time-driven routine that is periodically called by the local operating system and the routines that provide access to shared data. The platform-independent part consists of the list of running timers and routines that maintain that list. The list of running timers is implemented as a traditional delta list (the timer at the head of the list contains the absolute time interval whereas all other timers contain the time interval relative to the previous timer in the list).

To simplify timer maintenance, the function *StopTimer()* does not analyze the current status of the given timer (already expired or still running)—it simply marks the timer as expired. If the timer was still running, it will remain in the list of running timers. When it expires, it is forwarded to the given mailbox and from there it is discarded by the function member *Get()* of the class *TMailBox*.

4.4.3 Writing FSM Library–Based Implementations

Normally, we start by deriving subclasses from the base class *FiniteStateMachine*. For each such subclass, we must define the following functions (see Section 6.8 for more details):

- *GetMessageInterface()*: This function returns the pointer to the particular message interface object.
- *SetDefaultHeader()*: This function sets the default message header parameters.
- *GetMbxId()*: This function returns the identification of the mailbox associated to this FSM type.
- *GetAutomata()*: This function returns the identification of this FSM type.
- *SetDefaultFSMData()*: This function sets default FSM data.
- *NoFreeInstances()*: This recovery function is called when the pool of objects is exhausted.
- *Initialize()*: This function initializes FSM-related data, including the state transition table.

We then write the main program, which typically follows these steps:

- Create an instance of the class *FSMSystem*.
- Initialize the real-time kernel.
- Set the system parameters.
- Register (add) all FSM objects with the instance of the class *FSMSystem*.
- Start the system by calling the function *Start()* (defined within the class *FSMSystem*).

4.5 Examples

This section includes two representative examples of FSM Library–based implementations. The first example is the implementation of an application

for reading Internet electronic mail. The second example shows an implementation of the SIP invite client transaction.

4.5.1 Example 1

This example demonstrates how an application for reading Internet electronic mail can be constructed. The application is actually an e-mail client that comprises the following three objects (see the general collaboration diagram in Figure 4.15):

- *user*: a user interface
- *pop3*: the implementation of the POP3 protocol (refer to the original RFC 1939, freely available on the Internet at www.ietf.org/rfc/rfc1939.txt)
- *channel*: responsible for the direct communication with the e-mail server over the TCP protocol

As shown in Figure 4.15, the objects *user*, *pop3*, and *channel* are the instances of the classes *UserAuto*, *ClAuto*, and *ChAuto*, respectively. The object *pop3* is the central object. On its left side is the object *user*, and on its right side is the object *channel*. The interaction between these objects is illustrated with three typical scenarios that are shown in Figures 4.16 through 4.18. Figure 4.16 shows a successful session during which all pending e-mails are received and saved as files on a mass storage device. The flow of events from the point of view of the object *pop3* is as follows:

- Triggered by the reception of the message *User_Check_Mail* from the left object, it sends the message *Cl_Connection_Request* to the right object.
- Upon the reception of the message *Cl_Connection_Accept* from the right object, it sends the message *User_Connected* to the left object. The connection with the e-mail server is successfully established at this point.
- After receiving the username and password carried by the message *User_Name_Password* from the left object, it first sends the username in the message *MSG(USER name)* to the right object, which is acknowledged with the message *MSG(+OK)* from the right object, and it then sends the password in the message *MSG(PASS password)* to the right object, which is also acknowledged with the message *MSG(+OK)* from the right object. The user authentication procedure is successfully finished at this point.

FIGURE 4.15
Receive e-mail application collaboration diagram.

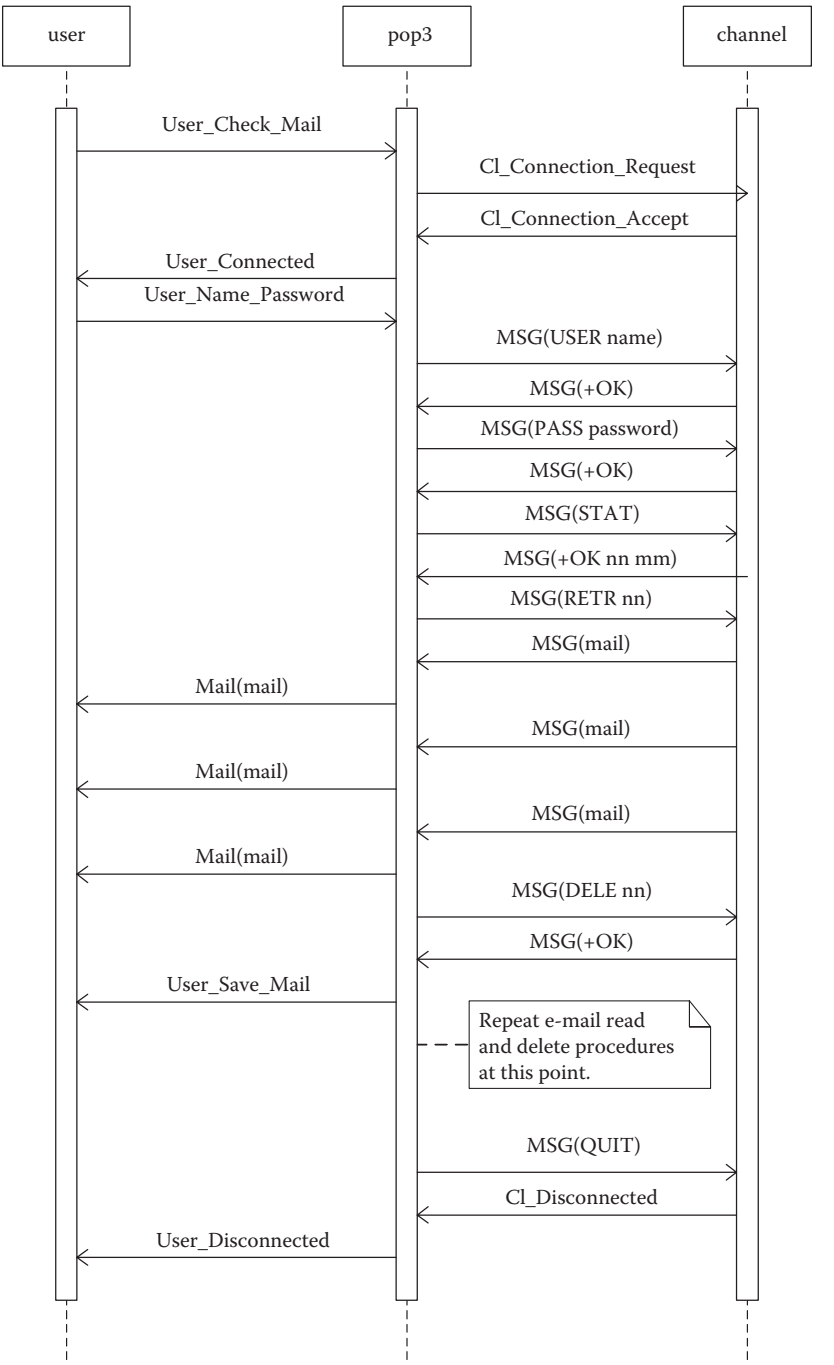


FIGURE 4.16
Successful receive e-mail session establishment scenario.

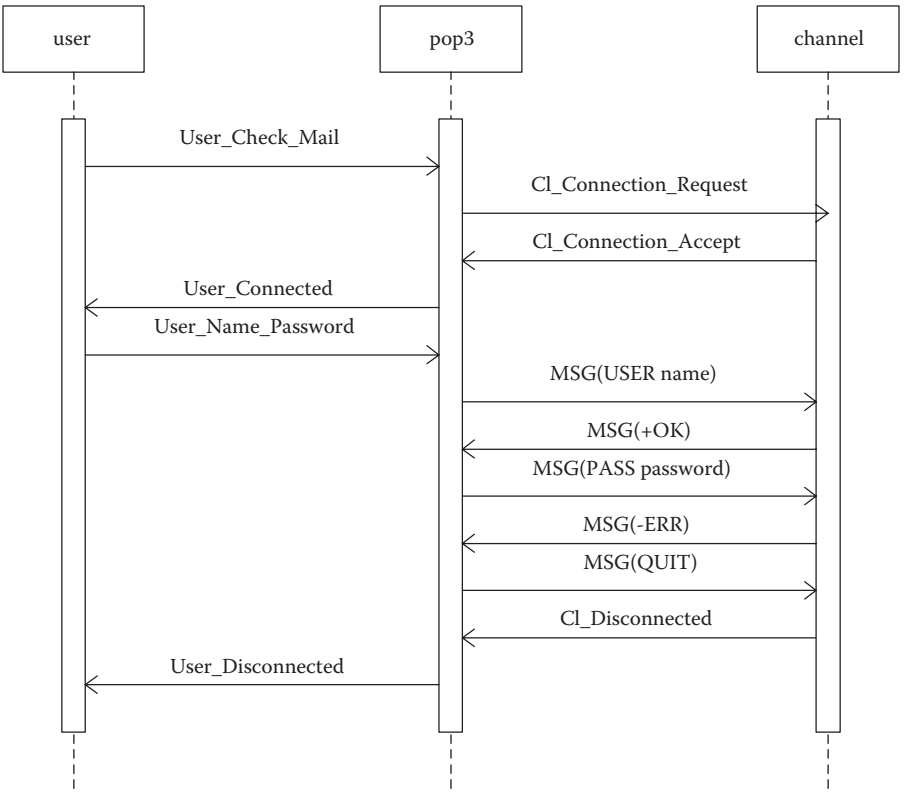


FIGURE 4.17
Invalid e-mail password processing scenario.

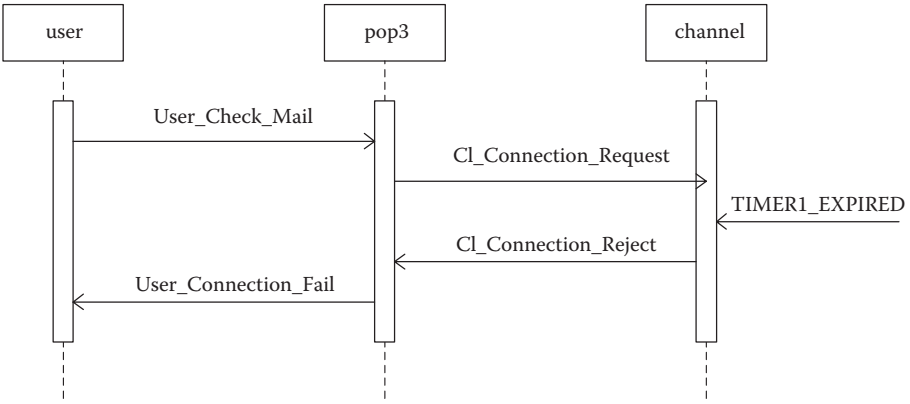


FIGURE 4.18
Unsuccessful receive e-mail session establishment scenario.

- It then checks the status of the pending e-mails by sending the message *MSG(STAT)* to the right object and receiving the answer in the message *MSG(+OK nn mm)*, where *nn* is the number of messages in the maildrop and *mm* is the size of the maildrop in octets.
- While pending e-mails remain, it repeats the sequence of the e-mail read procedure and the e-mail delete procedure. The e-mail read procedure starts with the message *MSG(RETR nn)* to the right object (*nn* is the order number of the e-mail message to be received). The right object, in turn, sends an e-mail message in a series of *MSG(mail)* messages (the size of the last one is smaller than 255 octets). The e-mail delete procedure starts with the message *MSG(DELE nn)* sent to the right object (*nn* is the order number of the message to be deleted by the e-mail server). After reception of the acknowledgment *MSG(+OK)* from the right object, the left object is informed accordingly with the message *User_Save_Mail* (normally, the object user should save the current e-mail message as a file on a mass storage device at this point).
- Finally, the object *pop3* starts the session closing procedure by sending the message *MSG(QUIT)* to the right object. Then, upon reception of the message *Cl_Disconnected* from the right object, it sends the message *User_Disconnected* to the left object.

Figure 4.17 shows the invalid password processing scenario. It is the same as the previous scenario up to the point where the object *pop3* sends the message *MSG(PASS password)* to the right object. Because the password is invalid, the right object responds with the message *MSG(-ERR)* and the object *pop3* immediately proceeds to the session closing procedure.

Figure 4.18 shows the unsuccessful session establishment scenario. It starts in the same way as the scenario in Figure 4.16. Assume that the TCP connection with the e-mail server cannot be established for some reason. Therefore, the *TIMER1_ID* that was started by the right object expires and the associate message *TIMER1_EXPIRED* triggers the right object to send the message *Cl_Connection_Reject*. The object *pop3*, in turn, sends the message *User_Connection_Fail* to the left object.

To keep this example simple enough, we focus further on the design and implementation of the key object in this application, the object *pop3*. The complete dynamic behavior of this object is specified with the SDL diagram, which is shown in Figures 4.19 and 4.20. The corresponding FSM is defined with nine states (*Cl_Ready*, *Cl_Connecting*, *Cl_Authorizing*, *Cl_User_Check*, *Cl_Pass_Check*, *Cl_Mail_Check*, *Cl_Receiving*, *Cl_Deleting*, and *Cl_Disconnecting*), six input messages (*User_Check_Mail*, *Cl_Connection_Reject*, *Cl_Connection_Accept*, *User_Name_Password*, *MSG*, and *Cl_Disconnected*), and seven output messages (*Cl_Connection_Request*, *User_Connection_Fail*, *User_Connected*, *MSG*, *Mail*, *User_Save_Mail*, and *User_Disconnected*).

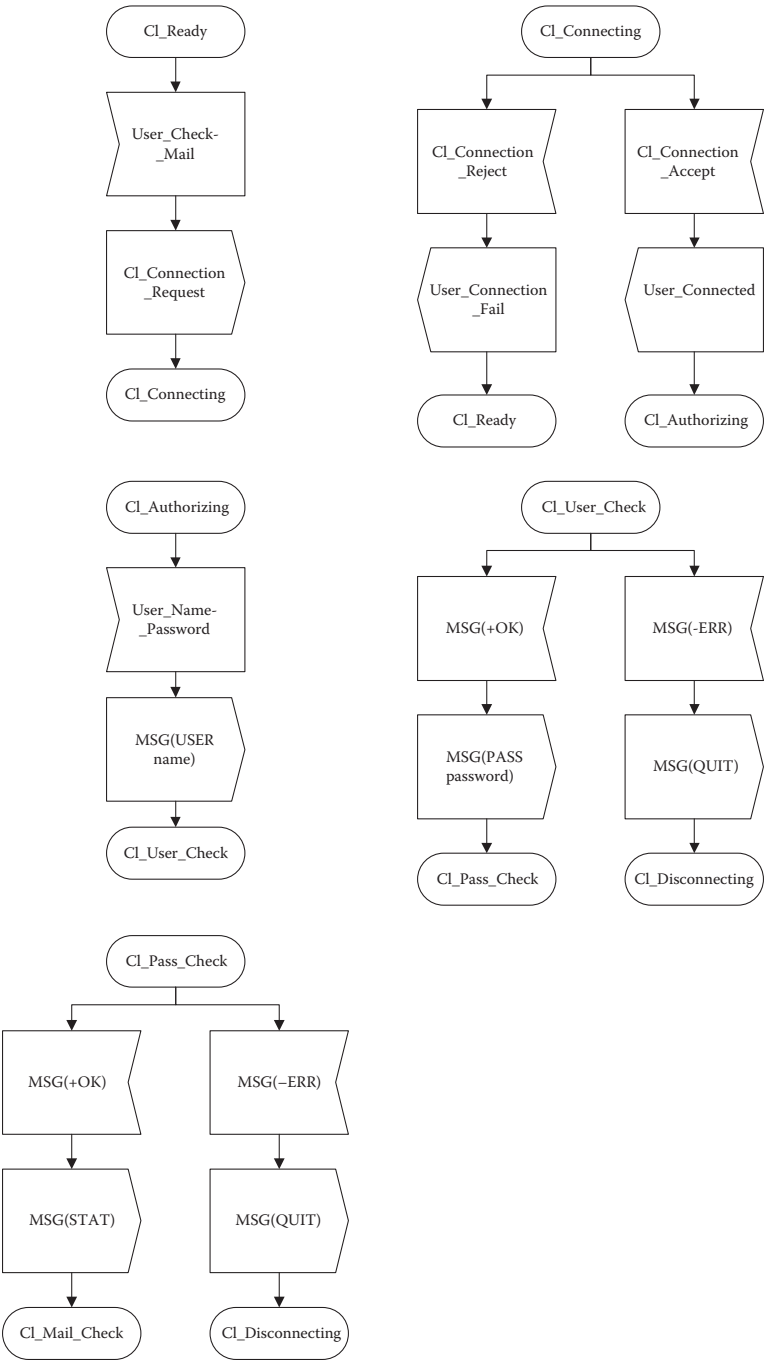


FIGURE 4.19
POP3 client SDL diagram, part I.

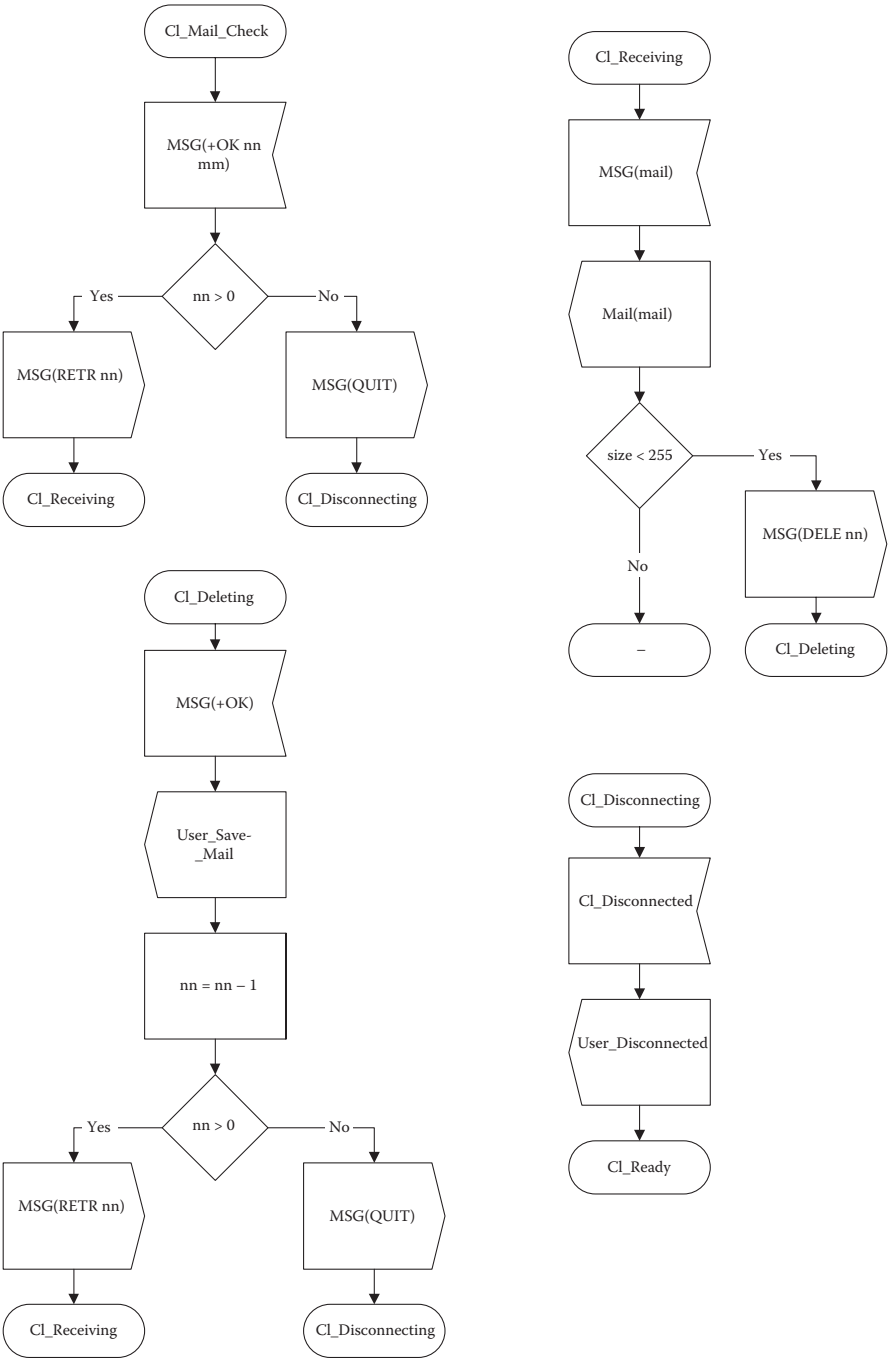


FIGURE 4.20
POP3 client SDL diagram, part II.

By convention, the names of all messages (except *Mail*) exchanged between the object *pop3* and the left object begin with the prefix *User_*. The names of the control messages exchanged between the object *pop3* and the right object begin with the prefix *Cl_*. The names of the POP3-related messages exchanged between the object *pop3* and the right object are named *MSG*. Two types of *MSG* messages are used—commands directed to the e-mail server and responses received from it.

The *MSG* commands are as follows:

- *MSG(USER name)* corresponds to the original POP3 command for specifying the name of the user mailbox.
- *MSG(PASS password)* corresponds to the original POP3 command for specifying the password for the previously specified mailbox.
- *MSG(STAT)* corresponds to the original POP3 command for inquiring about the mailbox status.
- *MSG(RETR nn)* corresponds to the original POP3 command for reading the pending e-mail message number *nn*.
- *MSG(DELE nn)* corresponds to the original POP3 command for deleting the pending e-mail message number *nn*.
- *MSG(QUIT)* corresponds to the original POP3 command for closing the current session.

The *MSG* responses are the following:

- *MSG(+OK)* corresponds to the original POP3 acknowledgment message.
- *MSG(ERR)* corresponds to the original POP3 error message.
- *MSG(mail)* corresponds to the actual e-mail message that was received from the e-mail server.

Figure 4.19 shows valid state transitions for the states *Cl_Ready*, *Cl_Connecting*, *Cl_Authorizing*, *Cl_User_Check*, and *Cl_Pass_Check*. The eight state transitions are shown in Figure 4.19, as follows:

- From *Cl_Ready* to *Cl_Connecting*, triggered by *User_Check_Mail*
- From *Cl_Connecting* to *Cl_Ready*, triggered by *Cl_Connection_Reject*
- From *Cl_Connecting* to *Cl_Authorizing*, triggered by *Cl_Connection_Accepted*
- From *Cl_Authorizing* to *Cl_User_Check*, triggered by *User_Name_Password*
- From *Cl_User_Check* to *Cl_Pass_check*, triggered by *MSG(+OK)*
- From *Cl_User_Check* to *Cl_Disconnecting*, triggered by *MSG(ERR)*

- From *Cl_Pass_Check* to *Cl_Mail_check*, triggered by *MSG(+OK)*
- From *Cl_Pass_Check* to *Cl_Disconnecting*, triggered by the *MSG(ERR)*

Figure 4.20 shows valid state transitions for the states *Cl_Mail_Check*, *Cl_Receiving*, *Cl_Deleting*, and *Cl_Disconnecting*. The seven state transitions are shown in Figure 4.20, as follows:

- From *Cl_Mail_Check* to *Cl_Receiving*, triggered by *MSG(+OK)* and guarded by the condition *nn* > 0
- From *Cl_Mail_Check* to *Cl_Disconnecting*, triggered by *MSG(+OK)* and guarded by the condition !(*nn* > 0)
- From *Cl_Receiving* to *Cl_Deleting*, triggered by *MSG(mail)* and guarded by the condition *mail(size)* < 255
- From *Cl_Receiving* to *Cl_Receiving*, triggered by *MSG(mail)* and guarded by the condition !(*mail(size)* < 255)
- From *Cl_Deleting* to *Cl_Receiving*, triggered by *MSG(+OK)* and guarded by the condition *nn* > 0
- From *Cl_Deleting* to *Cl_Disconnecting*, triggered by *MSG(+OK)* and guarded by the condition !(*nn* > 0)
- From *Cl_Disconnecting* to *Cl_Ready*, triggered by *Cl_Disconnected*

Next, we proceed to the implementation in C++ based on the FSM Library. First, we define symbolic constants specific for this project in a header file, which is typically named *const.h*. The content of this file is as follows:

```
#ifndef _CONST_H_
#define _CONST_H_
#include <fsm.h>
const uint8 CH_AUTOMATA_TYPE_ID = 0x00;
const uint8 CL_AUTOMATA_TYPE_ID = 0x01;
const uint8 USER_AUTOMATA_TYPE_ID = 0x02;

const uint8 CH_AUTOMATA_MBX_ID = 0x00;
const uint8 CL_AUTOMATA_MBX_ID = 0x01;
const uint8 USER_AUTOMATA_MBX_ID = 0x02;

// channel messages
const uint16 MSG_Connection_Request = 0x0001;
const uint16 MSG_Sock_Connection_Reject = 0x0002;
const uint16 MSG_Sock_Connection_Accept = 0x0003;
const uint16 MSG_Cl_MSG = 0x0004;
const uint16 MSG_Sock_MSG = 0x0005;
const uint16 MSG_Disconnect_Request = 0x0006;
const uint16 MSG_Sock_Disconnected = 0x0007;
const uint16 MSG_Sock_Disconnecting_Conf = 0x0008;

// pop3 client messages
const uint16 MSG_User_Check_Mail = 0x0009;
const uint16 MSG_Cl_Connection_Reject = 0x000a;
const uint16 MSG_Cl_Connection_Accept = 0x000b;
const uint16 MSG_User_Name_Password = 0x000c;
```

```
const uint16 MSG_MSG = 0x000d;
const uint16 MSG_Cl_Disconnected = 0x000f;

// user messages
const uint16 MSG_Set_All = 0x0010;
const uint16 MSG_User_Connected = 0x0011;
const uint16 MSG_User_Connection_Fail = 0x0012;
const uint16 MSG_Mail = 0x0013;
const uint16 MSG_User_Save_Mail = 0x0015;
const uint16 MSG_User_Disconnected = 0x0014;

#define ADRESS "krtlab8"
#define PORT 110

#define TIMER1_ID 1
#define TIMER1_COUNT 10
#define TIMER1_EXPIRED 0x20

#define PARAM_DATA 0x01
#define PARAM_Name 0x02
#define PARAM_Pass 0x03
#endif // _CONST_H_
```

The file *const.h* starts with the definitions of automata types and their private mailbox identifications. The identifications assigned to the classes *ChAuto*, *ClAuto*, and *UserAuto* are *CH_AUTOMATA_TYPE_ID*, *CL_AUTOMATA_TYPE_ID*, and *USER_AUTOMATA_TYPE_ID*, respectively. The identifications of their private mailboxes are *CH_AUTOMATA_MBX_ID*, *CL_AUTOMATA_MBX_ID*, and *USER_AUTOMATA_MBX_ID*, respectively. Next, we define the symbols that correspond to the codes of the messages recognized by the classes *ChAuto*, *ClAuto*, and *UserAuto*, respectively. By convention, these symbols are provided by prefixing the names of the messages from the SDL diagram (Figures 4.19 and 4.20) with the prefix *MSG_*.

At the end of the file *const.h*, we define the domain name and the number of the port, which are used to establish the TCP connection with the e-mail server (symbols *ADDRESS* and *PORT*), channel timer-related constants (symbols *TIMER1_ID*, *TIMER1_COUNT*, and *TIMER1_EXPIRED*), and the identifications of the message parameters (symbols *PARAM_DATA*, *PARAM_Name*, and *PARAM_Pass*).

Next, we write the header file *ClAuto.h*. Its content is as follows:

```
#ifndef _Cl_AUTO_H_
#define _Cl_AUTO_H_
#include <NetFSM.h>
#include <fsmsystem.h>
#include "const.h"
class ClAuto : public FiniteStateMachine {
 // for FSM
 StandardMessage StandardMsgCoding;
 MessageInterface *GetMessageInterface(uint32 id);
 void SetDefaultHeader(uint8 infoCoding);
 void SetDefaultFSMData();
 void NoFreeInstances();
 void Reset();
 uint8 GetMbxId();
```

```
 uint8 GetAutomata();
 uint32 GetObject();
 void ResetData();
 // FSM States
 enum ClStates {
  FSM_Cl_Ready,
  FSM_Cl_Connecting,
  FSM_Cl_Authorizing,
  FSM_Cl_User_Check,
  FSM_Cl_Pass_Check,
  FSM_Cl_Mail_Check,
  FSM_Cl_Receiving,
  FSM_Cl_Deleting,
  FSM_Cl_Disconnecting
 };
public:
 ClAuto();
 ~ClAuto();
 void Initialize();
 void FSM_Cl_Ready_User_Check_Mail();
 void FSM_Cl_Connecting_Cl_Connection_Reject();
 void FSM_Cl_Connecting_Cl_Connection_Accept();
 void FSM_Cl_Authorizing_User_Name_Password();
 void FSM_Cl_User_Check_MSG();
 void FSM_Cl_Pass_Check_MSG();
 void FSM_Cl_Mail_Check_MSG();
 void FSM_Cl_Receiving_MSG();
 void FSM_Cl_Deleting_MSG();
 void FSM_Cl_Disconnecting_Cl_Disconnected();
protected:
 int m_MessageCount;
 char m_UserName[20];
 char m_Password[20];
};
#endif /* _Cl_AUTO_H */
```

After listing all necessary header files, we declare the class *ClAuto*, which is derived from the base class *FiniteStateMachine*. The declaration of the class *ClAuto* starts with the declaration of field and function members that are mandatory for any class that is derived from the class *FiniteStateMachine* (as explained previously in this chapter). It continues with the declaration of FSM state names and state transition function prototypes.

By convention, FSM state names are the names from the SDL diagram with the prefix *FSM_* (e.g., the initial state *Cl_Ready* is named *FSM_Cl_Ready* in the C++ code). The state transition function is named by concatenating the state name and the input message name and by prefixing this composite name with *FSM_* (e.g., the state transition function performed when the FSM in state *Cl_Ready* receives the message *User_Check_Mail* is named *FSM_Cl_Ready_User_Check_Mail*). As previously mentioned, *ClAuto* FSM has nine states and fourteen state transitions.

The reader may be puzzled with the fact that there are fourteen valid FSM state transitions and only ten state transition functions declared in the header file *ClAuto.h*. This circumstance is because some of the state transitions are triggered with the same message type but different message content—e.g., *MSG(+OK)* and *MSG(–ERR)*—or they are guarded with the complementary

conditions—e.g., (*nn* > 0) and !(*nn* > 0). To clearly understand these matters, remember that *FiniteStateMachine* derivatives react to various message types in various FSM states. This is how we calculate the number of state transitions.

If we apply the principle stated above to the class *ClAuto*, we have the situation where all the states react to a single message with the exception of the state *Cl_Connecting*, which reacts to two valid messages, *Cl_Connection_Reject* and *Cl_Connection_Accept*. Because of this, we have (8 × 1) + (1 × 2) state transition functions, which resolves to ten state transition functions, as mentioned above.

Finally, we write the class *ClAuto* definition file, named *ClAuto.cpp*. The content of this file is as follows:

```
#include <stdio.h>
#include "const.h"
#include "ClAuto.h"
#define StandardMessageCoding 0x00

ClAuto::ClAuto() : FiniteStateMachine(0, 9, 2) {}
ClAuto::~ClAuto() {}

uint8 ClAuto::GetAutomate() {
 return CL_AUTOMATA_TYPE_ID;
}

uint8 ClAuto::GetMbxId() {
 return CL_AUTOMATA_MBX_ID;
}

uint32 ClAuto::GetObject() {
 return GetObjectId();
}

MessageInterface *ClAuto::GetMessageInterface(uint32 id) {
 return &StandardMsgCoding;
}

void ClAuto::SetDefaultHeader(uint8 infoCoding) {
 SetMsgInfoCoding(infoCoding);
 SetMessageFromData();
}

void ClAuto::SetDefaultFSMData() {
 SetDefaultHeader(StandardMessageCoding);
}

void ClAuto::NoFreeInstances() {
 printf("[%d] ClAuto::NoFreeInstances()\n",  GetObjectId());
}

void ClAuto::Reset() {
 printf("[%d] ClAuto::Reset()\n", GetObjectId());
}

void ClAuto::Initialize() {
 SetState(FSM_Cl_Ready);

 // set message handlers
 InitEventProc(FSM_Cl_Ready, MSG_User_Check_Mail,
(PROC_FUN_PTR)&ClAuto::FSM_Cl_Ready_User_Check_Mail));
```

```
 InitEventProc(FSM_Cl_Connecting,  MSG_Cl_Connection_Reject,

(PROC_FUN_PTR)&ClAuto::FSM_Cl_Connecting_Cl_Connection_Reject));

 InitEventProc(FSM_Cl_Connecting,  MSG_Cl_Connection_Accept,

(PROC_FUN_PTR)&ClAuto::FSM_Cl_Connecting_Cl_Connection_Accept));

 InitEventProc(FSM_Cl_Authorizing,  MSG_User_Name_Password,

(PROC_FUN_PTR)&ClAuto::FSM_Cl_Authorizing_User_Name_Password));

 InitEventProc(FSM_Cl_User_Check, MSG_MSG,
  (PROC_FUN_PTR)&ClAuto::FSM_Cl_User_Check_MSG));

 InitEventProc(FSM_Cl_Pass_Check, MSG_MSG,
  (PROC_FUN_PTR)&ClAuto::FSM_Cl_Pass_Check_MSG));

 InitEventProc(FSM_Cl_Mail_Check, MSG_MSG,
  (PROC_FUN_PTR)&ClAuto::FSM_Cl_Mail_Check_MSG));

 InitEventProc(FSM_Cl_Receiving, MSG_MSG,
  (PROC_FUN_PTR)&ClAuto::FSM_Cl_Receiving_MSG));

 InitEventProc(FSM_Cl_Deleting, MSG_MSG,
  (PROC_FUN_PTR)&ClAuto::FSM_Cl_Deleting_MSG));

 InitEventProc(FSM_Cl_Disconnecting,  MSG_Cl_Disconnected,
(PROC_FUN_PTR)&ClAuto::FSM_Cl_Disconnecting_Cl_Disconnected));
}

void ClAuto::FSM_Cl_Ready_User_Check_Mail(){
 PrepareNewMessage(0x00, MSG_Connection_Request);
 SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
 SetMsgObjectNumberTo(0);
 SendMessage(CH_AUTOMATA_MBX_ID);
 SetState(FSM_Cl_Connecting);
}

void ClAuto::FSM_Cl_Connecting_Cl_Connection_Reject(){
 PrepareNewMessage(0x00, MSG_User_Connection_Fail);
 SetMsgToAutomata(USER_AUTOMATA_TYPE_ID);
 SetMsgObjectNumberTo(0);
 SendMessage(USER_AUTOMATA_MBX_ID);
 SetState(FSM_Cl_Ready);
}

void ClAuto::FSM_Cl_Connecting_Cl_Connection_Accept(){
 PrepareNewMessage(0x00, MSG_User_Connected);
 SetMsgToAutomata(USER_AUTOMATE_TYPA_ID);
 SetMsgObjectNumberTo(0);
 SendMessage(USER_AUTOMATA_MBX_ID);
 SetState(FSM_Cl_Authorizing);
}

void ClAuto::FSM_Cl_Authorizing_User_Name_Password(){
 char* name = new char[20];
 char* pass = new char[20];
 uint8* buffer = GetParam(PARAM_Name);

 memcpy(m_UserName,buffer+2,buffer[1]);
 m_UserName[buffer[1]] = 0;     // terminate string
 buffer = GetParam(PARAM_Pass);
```

```
 memcpy(m_Password,buffer+2,buffer[1]);
 m_Password[buffer[1]] = 0;    // terminate string
 char l_Command[20] = "user";
 strcpy(l_Command+5,m_UserName);
 strcpy(l_Command+5+strlen(m_UserName),"\r\n");

 PrepareNewMessage(0x00, MSG_Cl_MSG);
 SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
 SetMsgObjectNumberTo(0);
AddParam(PARAM_DATA,strlen(l_Command),(uint8*)l_Command);
 SendMessage(CH_AUTOMATA_MBX_ID);
 SetState(FSM_Cl_User_Check);
}

void ClAuto::FSM_Cl_User_Check_MSG(){
 char* data = new char[255];
 uint8* buffer = GetParam(PARAM_DATA);
 uint16 size = buffer[1];

 memcpy(data,buffer + 2,size);
 data[size]=0;
 printf("%s",data);
 if((data[0] == '+')) {
  char l_Command[20] = "pass ";
  strcpy(l_Command+5,m_Password);
  strcpy(l_Command+5+strlen(m_Password),"\r\n");
  PrepareNewMessage(0x00, MSG_Cl_MSG);
  SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
  SetMsgObjectNumberTo(0);
AddParam(PARAM_DATA,strlen(l_Command),(uint8*)l_Command);
  SendMessage(CH_AUTOMATA_MBX_ID);
  SetState(FSM_Cl_Pass_Check);
  else {
  char l_Command[20] = "quit\r\n";
  PrepareNewMessage(0x00, MSG_Cl_MSG);
  SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
  SetMsgObjectNumberTo(0);
  AddParam(PARAM_DATA,6,(uint8*)l_Command);
  SendMessage(CH_AUTOMATA_MBX_ID);
  SetState(FSM_Cl_Disconnecting);
 }
}

void ClAuto::FSM_Cl_Pass_Check_MSG(){
 char* data = new char[255];
 uint8* buffer = GetParam(PARAM_DATA);
 uint16 size = buffer[1];

 memcpy(data,buffer + 2,size);
 data[size]=0;
 printf("%s",data);
 if((data[0] == '+')) {
  char l_Command[20] = "stat\r\n";
  PrepareNewMessage(0x00, MSG_Cl_MSG);
  SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
  SetMsgObjectNumberTo(0);
  AddParam(PARAM_DATA,6,(uint8*)l_Command);
  SendMessage(CH_AUTOMATA_MBX_ID);
  SetState(FSM_Cl_Mail_Check);
  else {
  char l_Command[20] = "quit\r\n";
  PrepareNewMessage(0x00, MSG_Cl_MSG);
  SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
  SetMsgObjectNumberTo(0);
```

```
  AddParam(PARAM_DATA,6,(uint8*)l_Command);
  SendMessage(CH_AUTOMATA_MBX_ID);
  SetState(FSM_Cl_Disconnecting);
 }
}

void ClAuto::FSM_Cl_Mail_Check_MSG(){
 char* data = new char[255];
 uint8* buffer = GetParam(PARAM_DATA);
 uint16 size = buffer[1];

 memcpy(data,buffer+2,size);
 data[size]=0;
 printf("%s",data);
 int l_nDigit = 1;
 while(buffer[l_nDigit+6] != ' ') l_nDigit++;
 memcpy(data,buffer +6,l_nDigit);
 data[l_nDigit]=0;
 m_MessageCount = atoi(data);

 if((m_MessageCount == 0) {
  char l_Command[20] = "quit\r\n";
  PrepareNewMessage(0x00, MSG_Cl_MSG);
  SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
  SetMsgObjectNumberTo(0);
  AddParam(PARAM_DATA,6,(uint8*)l_Command);
  SendMessage(CH_AUTOMATA_MBX_ID);
  SetState(FSM_Cl_Disconnecting);
  else {
  char l_Command[20] = "retr ";
  strcpy(l_Command+5,data);
  strcpy(l_Command+5+l_nDigit,"\r\n");
  PrepareNewMessage(0x00, MSG_Cl_MSG);
  SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
 SetMsgObjectNumberTo(0);

AddParam(PARAM_DATA,5+l_nDigit+2,(uint8*)l_Command);
  SendMessage(CH_AUTOMATA_MBX_ID);
  SetState(FSM_Cl_Receiving);
 }
}

void ClAuto::FSM_Cl_Receiving_MSG(){
 char* data = new char[255];
 uint8* buffer = GetParam(PARAM_DATA);
 uint16 size = buffer[1];

 memcpy(data,buffer + 2,size);
 char temp[4];
 memcpy(temp,data,3); temp[3] = 0;
 if((strcmp(temp,"+OK") != 0) {
  PrepareNewMessage(0x00, MSG_Mail);
  SetMsgToAutomata(USER_AUTOMATA_TYPE_ID);
  SetMsgObjectNumberTo(0);
  AddParam(PARAM_DATA,size,(uint8*)data);
  SendMessage(USER_AUTOMATA_MBX_ID);
  if((size < 255) {
   char l_Command[20] = "dele ";
   itoa(m_MessageCount,data,10);
   strcpy(l_Command+5,data);
   strcpy(l_Command+5+strlen(data),"\r\n");
   PrepareNewMessage(0x00, MSG_Cl_MSG);
   SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
   SetMsgObjectNumberTo(0);
```

```
AddParam(PARAM_DATA,5+strlen(data)+2,(uint8*)l_Command);
   SendMessage(CH_AUTOMATA_MBX_ID);
   SetState(FSM_Cl_Deleting);
  }
 }
}

void ClAuto::FSM_Cl_Deleting_MSG(){
 PrepareNewMessage(0x00, MSG_User_Save_Mail);
SetMsgToAutomata(USER_AUTOMATA_TYPE_ID);
 SetMsgObjectNumberTo(0);
 SendMessage(USER_AUTOMATA_MBX_ID);
 m_MessageCount--;
 if(m_MessageCount > 0) {
  char data[5];
  char l_Command[20] = "retr ";
  itoa(m_MessageCount,data,10);
  strcpy(l_Command+5,data);
  strcpy(l_Command+5+strlen(data),"\r\n");
  PrepareNewMessage(0x00, MSG_Cl_MSG);
  SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
  SetMsgObjectNumberTo(0);

AddParam(PARAM_DATA,5+strlen(data)+2,(uint8*)l_Command);
  SendMessage(CH_AUTOMATA_MBX_ID);
  SetState(FSM_Cl_Receiving);
  else {
  char l_Command[20] = "quit\r\n";
  PrepareNewMessage(0x00, MSG_Cl_MSG);
  SetMsgToAutomata(CH_AUTOMATA_TYPE_ID);
  SetMsgObjectNumberTo(0);
  AddParam(PARAM_DATA,6,(uint8*)l_Command);
  SendMessage(CH_AUTOMATA_MBX_ID);
  SetState(FSM_Cl_Disconnecting);
 }
}

void ClAuto::FSM_Cl_Disconnecting_Cl_Disconnected(){
 PrepareNewMessage(0x00, MSG_User_Disconnected);
 SetMsgToAutomata(USER_AUTOMATA_TYPE_ID);
 SetMsgObjectNumberTo(0);
 SendMessage(USER_AUTOMATA_MBX_ID);
 SetState(FSM_Cl_Ready);
}
```

The file *ClAuto.cpp* starts with a list of all necessary header files (*stdio.h*, *const.h*, and *ClAuto.h*), followed by the definition of the symbolic constant *StandardMessageCoding* and the set of mandatory function definitions: class constructor, class destructor, and functions *GetAutomata()*, *GetMbxId()*, *GetObject()*, *GetMessageInterface()*, *SetDefaultHeader()*, *SetDefaultFSMData()*, *NoFreeInstances()*, *Reset()*, and *Initialize()*.

The class constructor *ClAuto()* calls the constructor of the class *FiniteStateMachine* with a list of parameters, which specifies that the *ClAuto* FSM has no timers, nine states, and the maximum of two state transitions per state (see the FSM Library API specification in Section 6.8, particularly, Section 6.8.11). The class destructor performs no particular operation.

The mandatory functions provide the following functionalities:

- The function *GetAutomata()* returns the *ClAutomata* type identification (the constant *CL_AUTOMATA_TYPE_ID*). See also Section 6.8.24.
- The function *GetMbxId()* returns the associated mailbox identification (the constant *CL_AUTOMATA_MBX_ID*). See also Section 6.8.38.
- The function *GetObject()* returns the object identification. Actually, it returns the value returned by the FSM Library function *GetObject Id()*. See also Section 6.8.60.
- The function *GetMessageInterface()* returns the pointer to the message coding object (an instance of the class *StandardMessage*). See also Section 6.8.39.
- The function *SetDefaultHeader()* sets default data in the new message header by calling two FSM Library functions, *SetMsgInfoCoding()* and *SetMessageFromData()*. See also Section 6.8.97, Section 6.8.117, and Section 6.8.108.
- The function *SetDefaultFSMData()* sets the new message header default values by calling the function *SetDefaultHeader()* and specifying the constant *StandardMessageCoding* as its parameter.
- The function *NoFreeInstances()* just prints the information message to the standard output file. See also Section 6.8.78.
- The function *Reset()* also just prints the information message to the standard output file. See also Section 6.8.85.

The most important mandatory function is the function *Initialize()*. It starts by setting the FSM initial state, *Cl_Ready* (denoted with the constant *FSM_Cl_Ready*). It continues by setting the state transition functions (also referred to as message handlers). Each message handler is set by a single call to the FSM Library function *InitEventProc()*. The first parameter of this function is the state name, the second is the input message name, and the third is the address of the corresponding *ClAuto* function member (see also Section 6.8.73).

The set of mandatory functions is followed by the set of state transition functions. As already mentioned, ten such functions are used. Each of these functions processes a single message type in a single state, as follows:

- The function *FSM_Cl_Ready_User_Check_Mail()* processes the message *User_Check_Mail* in the state *Cl_Ready*.
- The function *FSM_Cl_Connecting_Cl_Connection_Reject()* processes the message *Cl_Connection_Reject in the state Cl_Connecting*.
- The function *FSM_Cl_Connecting_Cl_Connection_Accept()* processes the message *Cl_Connection_Accept* in the state *Cl_Connecting*.

- The function *FSM_Cl_Authorizing_User_Name_Password()* processes the message *User_Name_Password* in the state *Cl_Authorizing*.
- The function *FSM_Cl_User_Check_MSG()* processes the message *MSG* in the state *Cl_User_Check*.
- The function *FSM_Cl_Pass_Check_MSG()* processes the message *MSG* in the state *Cl_Pass_Check*.
- The function *FSM_Cl_Mail_Check_MSG()* processes the message *MSG* in the state *Cl_Mail_Check*.
- The function *FSM_Cl_Receiving_MSG()* processes the message *MSG* in the state *Cl_Receiving*.
- The function *FSM_Cl_Deleting_MSG()* processes the message *MSG* in the state *Cl_Deleting*.
- The function *FSM_Cl_Disconnecting_Cl_Disconnected()* processes the message *Cl_Disconnected* in the state *Cl_Disconnecting*.

The function *FSM_Cl_Ready_User_Check_Mail()* is a typical simple state transition function. First, it creates a new message by calling the function *PrepareNewMessage()*. (Its first parameter is the message length and the second is the message type; the third parameter is optional and is not used in this example. See also Section 6.8.81.) It then sets the destination FSM type and object identification by calling the function *SetMsgToAutomata()* (its parameter is the FSM type identification; see also Section 6.8.125) and the function *SetMsgObjectNumberTo()* (its parameter is the FSM object identification; see also Section 6.8.123), respectively. Next, it sends the new message to the destination mailbox by calling the function *SendMessage()* (its parameter is the mailbox identification; see also Section 6.8.106). Finally, it sets the new FSM state by calling the function *SetState* (its parameter is the state identification; see also Section 6.8.137).

The next two functions, *FSM_Cl_Connecting_Cl_Connection_Reject()* and *FSM_Cl_Connecting_Cl_Connection_Accept()*, are very similar to the one previously described (only the message type and the new state name are different). But the fourth state transition function, *FSM_Cl_Authorizing_User_Name_Password()*, is more complex. It demonstrates well how a state transition function can get a parameter from the current message and how it can add a parameter to the new message. This concrete state transition function gets two parameters (username and password) from the current message by calling the function *GetParam()* (its parameter is the identification of the parameter type; see also Section 6.8.61). It also adds one parameter (username) to the new message by calling the function *AddParam()* (its parameters are the message parameter type, length, and pointer; see also Section 6.8.12).

The fifth state transition function, *FSM_Cl_User_Check_MSG()*, is even more complex because it involves branching depending on the value of the current message parameter. By making a branch, the state transition function

actually selects one of two possible paths of the FSM evolution, which yields two different output (new) messages and two different destination FSM states. The sixth state transition function is very similar to the fifth one.

The seventh state transition function, *FSM_Cl_Mail_Check_MSG()*, brings one new important detail. It shows how a state transition function can save some data (in this example, the number of pending e-mail messages, which is stored in the class field member *m_MessageCount*) so that it can be shared or used by other state transitions—in this example, by the ninth state transition function, *FSM_Cl_Deleting_MSG()*.

The rest of the state transition functions do not bring anything essentially new. However, the reader is advised to study them in detail as an additional exercise.

4.5.2 Example 2

The aim of this example is to implement the SIP invite client transaction design, which is given in Section 3.10.5 (Chapter 3, Example 5). Briefly, in that section we examined the general collaboration diagram of the SIP Softphone (see Section 2.3.3, Figure 2.16) with the focus on the invite client transaction. The result is the general collaboration diagram shown in Figure 3.69. We then made two particular collaboration diagrams and their semantically equivalent sequence diagrams for the cases of successful and unsuccessful SIP session establishment (Figures 3.70 through 3.73). Finally, we devised the complete dynamic behavior specification in the form of the statechart diagram (Figure 3.74) and semantically equivalent SDL diagram (Figures 3.75 through 3.78).

We start the implementation of this design by defining the symbolic constants, such as the FSM type names (e.g., the name of the invite client FSM type is *InviteClienteTE_FSM*), mailbox names (e.g., the name of the invite client mailbox is *InviteClienteTE_FSM_MBX*), names of the FSM Library related message types, timer names (e.g., *TIMER_A*, *TIMER_B*, and *TIMER_D*), names of the SIP messages (e.g., *INVITE*, *OPTIONS*, *CANCEL*, *ACK*, *BYE*, and *RESISTER*), names of the response codes (e.g., *_180_RINGING*, *_200_OK*, *_302_MOVED_TEMPORARILY*, *_401_UNAUTHORIZED*, *_403_FORBIDDEN*, and *_404_NOT_FOUND*), and names of situations (e.g., *URI_IN_TO_UNRECOGNIZED* and *NOT_TO_CURRENT_USER*). Traditionally, we write definitions of all these constants in the file *constants.h*.

Next, we write the class that represents an SIP message, simply named *Message*. The most important field member of this class is the last (also referred to as the current) SIP message (its type is the C++ type string). Other field members hold the relevant SIP session related information. The function members support SIP message analysis and synthesis (parsing and creation). Actually, the class *Message* that is used in this example is a simple wrapper around the OpenSIP SIP message parser. (OpenSIP is freely available on the Internet at https://www.opensips.org/.)

We skip the content of the file *constants.h* and the source code of the class *Message* intentionally to keep this example short enough and easily comprehendible, and we proceed with the introduction of the supplementary class *TALE*. The declaration of the class *TALE* is as follows:

```
#ifndef _TALE_FSM_
#define _TALE_FSM_
#include "../kernel/fsm.h"
#include "../message/message.h"
#include "../constants.h"

class TALE : public FiniteStateMachine {
 uint8 MessageCopy[MAX_LENGTH_MESSAGE];
 uint32 IndexTLI;
 BOOL IndexTLISet;
public:
 void SetIndexTLI(uint32 newIndexTLI);
 uint32 GetIndexTLI();
 BOOL IsTransportReliable();
 void SendMessageToTU();
 void SendMessageToTPL();
 void SendErrorMessageToTU();
 void MakeLocalCopyOfMsg();
 void SendCopiedMessageToTPL();

public:
 TALE(uint16 numOfTimers, uint16 numOfState, uint16 maxNumOfPrPerSt);
 ~TALE();
};
```

The class *TALE* is a good example of how we can make our implementations more compact. As we can see from the previous example, sending a single message requires a series of FSM Library function calls. For example, forwarding the current message would require a series of calls to the function *CopyMessage()*, *SetMsgToAutomata()*, *SetMsgToGroup()*, *SetMsgObjectNumberTo()*, and function *SendMessage()*—five function calls. In the case of simple designs, we can tolerate repetition of this series of function calls, but in cases requiring more complex design or platforms with limited resources, this repetition may not be tolerated.

Consider the SIP invite client transaction FSM. It has thirteen state transitions, and most of them require sending a message to either the TPL (transport layer) or TU (transaction user). We would need to repeat the same series of function calls about ten times. Consider now the whole SIP Softphone, which supports four types of transactions (invite, non-invite, client, and server transactions). In such situations, replacing this series of function calls with a single function call (which, in turn, performs the original sequence of function calls) makes sense.

This replacement is exactly the reason why the class *TALE* has been introduced in the first place. This class inherits all field and function members from the class *FiniteStateMachine*, from which it is derived. It also adds some new field and function class members. All classes that implement SIP transactions are derived from the class *TALE*. The most important field member of the class *TALE* is the field *MessageCopy*, which holds the copy of the last sent

message. Actually, this field is the retransmission buffer (remember that SIP invite client in the state *Calling* must retransmit the message *INVITE* in case the timer A expires).

The two most important function members are the functions *SendMessageToTU()* and *SendMessageToTPL()*. The former sends the current message to TU and the latter to TPL. They are very similar; therefore, it is sufficient to study just one of them. Here is the source code of the former function:

```
void TALE::SendMessageToTU() {
 CopyMessage();
 SetMsgToAutomata(UA_Disp_FSM);
 SetMsgToGroup(INVALID_08);
 SetMsgObjectNumberTo(0);
 SendMessage(UA_Disp_FSM_MBX);
}
```

This is the most elegant way to forward a message in FSM Library-based implementations. The function *CopyMessage()* copies the current (last received) message to the new (output) message. The symbolic constant *UA_Disp_FSM* is the name of the UA (user agent) FSM type, and the constant *UA_Disp_FSM_MBX* is the name of its mailbox. As we will shortly see, the use of the functions *SendMessageToTU()* and *SendMessageToTPL()* significantly compresses the source code. They make one-to-one mapping of SDL diagrams to C++ code possible.

Next, we proceed to the implementation of the invite client transaction FSM. We implement it by writing the class *InviteClientTE*. Note that in Figures 3.69 through 3.73, we used the abbreviation *InClientT* for this name. The declaration of the class *InviteClientTE* is as follows:

```
#ifndef _InviteClientTE_FSM_
#define _InviteClientTE_FSM_
#include "TALE.h"

class InviteClientTE : public TALE {
 Message SIPMsg;
 uint32 cseq_number;
 uint32 TimerADuration;

public:
 enum States {
  STATE_INITIAL,
  STATE_CALLING,
  STATE_PROCEEDING,
  STATE_COMPLETED
 };
 // state Initial message handlers
 void Evt_Init_INVITE();
 // state Calling message handlers
 void Evt_Calng_TIMER_A_EXP();
 void Evt_Calng_RESPONSE_1XX();
 void Evt_Calng_RESPONSE_2XX();
 void Evt_Calng_TIMER_B_EXP();
 void Evt_Calng_RESPONSE_3_6XX();
 void Evt_Calng_TRANSPORT_ERR();
 // state Proceeding message handlers
```

```
 void Evt_Proc_RESPONSE_1XX();
 void Evt_Proc_RESPONSE_2XX();
 void Evt_Proc_RESPONSE_3_6XX();
 // state Completed message handlers
 void Evt_Comptd_TIMER_D_EXP();
 void Evt_Comptd_RESPONSE_3_6XX();
 void Evt_Comptd_TRANSPORT_ERR();
 // unexpected messages message handler
 void Event_UNEXPECTED();
 // problem specific functions
 void RetransmitInvite();
 BOOL SendAckMessageToTPL();
 // FiniteStateMachine abstract functions
 StandardMessage StandardMsgCoding;
 MessageInterface *GetMessageInterface(uint32 id);
 void SetDefaultHeader(uint8 infoCoding);
 void SetDefaultFSMData();
 void NoFreeInstances();
 void Reset();
 uint8 GetMbxId();
 uint8 GetAutomate();
 uint32 GetObject();
 void ResetData();
public:
```

The class *InviteClientTE* is derived from the class *TALE*. The meaning of its field members is as follows:

- The field *SIPMsg* is the SIP message parser (an instance of the class *Message*).
- The field *cseq_number* holds the value of the SIP message header field *CSeq*, which is used to identify and order transactions (see RFC 3261, Subsection 8.1.1.5).
- The field *TimerADuration* contains the current value of the timer A (remember, the value of the timer A is doubled each time it expires).

Next, we enumerate the names of the FSM states. There are altogether four FSM states: *STATE_INITIAL*, *STATE_CALLING*, *STATE_PROCEEDING*, and *STATE_COMPLETED*. A short explanation is needed at this point. According to the original specification (RFC 3261, Figure 5, page 128), the invite client transaction FSM also has four explicitly rendered states, namely, *Calling*, *Proceeding*, *Completed*, and *Terminated*. The initial state is omitted in the original specification. In our implementation, we create a pool of *InviteClientTE* objects, which are dynamically allocated on demand by the TU. These objects are never really terminated. Once they play their simple role, they are returned to the pool of free *InviteClientTE* objects, and from there they are dynamically assigned to play the same role again. Therefore, we renamed the state *Terminated* to *Initial*. We also made this state the source of the initial state transition (triggered with the message *INVITE* from TU), thus making the FSM a never-terminating one.

We then list the state transition function prototypes for each state individually. The naming convention is the same as in the previous example:

The name of the state transition function is constructed by concatenating the state name and the message name and by prefixing that name with a certain prefix. The naming convention is applied more freely in this example by shortening the state names. This practice is frequently done to keep the name lengths acceptable (short enough, but providing code readability at the same time). Thirteen valid state transitions and their corresponding state transition functions (message handlers) are used. The fourteenth message handler, named *Event_UNEXPECTED()*, handles all unexpected messages in all states.

Finally, we list the function prototypes of the problem-specific functions and mandatory *FiniteStateMachine* abstract functions. These functions—except the function *RetransmitInvite()*—are intentionally skipped in the text that follows to keep the presentation of this example short.

We finish the implementation by writing the class *InviteClientTE* definition file, named *InvClientTE.cpp*. The content of this file is as follows:

```
#include <stdio.h>
#include "InvClientTE.h"
#include "../Message/message.h"
#include "timer_values.h"
#define StandardMessageCoding 0x00

InviteClientTE::InviteClientTE() : TALE(10, 10, 10) {}
InviteClientTE::~InviteClientTE() {}

void InviteClientTE::Initialize() {
 SetState(STATE_INITIAL);
 // define timers
 InitTimerBlock(TIMER_A,1,TIMER_A_EXPIRED);
 InitTimerBlock(TIMER_B,1,TIMER_B_EXPIRED);
 InitTimerBlock(TIMER_D,1,TIMER_D_EXPIRED);
 // state STATE_INITIAL message handlers
 InitEventProc(STATE_INITIAL, INVITE,
  (PROC_FUN_PTR)&InviteClientTE::Evt_Init_INVITE);
 // state STATE_CALLING message handlers
InitEventProc(STATE_CALLING, TIMER_A_EXPIRED,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Calng_TIMER_A_EXP);

InitEventProc(STATE_CALLING, RESPONSE_1XX_T,
 (PRO_FUN_PTR)&InviteClientTE::Evt_Calng_RESPONSE_1XX);

InitEventProc(STATE_CALLING, RESPONSE_2XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Calng_RESPONSE_2XX);

InitEventProc(STATE_CALLING, TIMER_B_EXPIRED,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Calng_TIMER_B_EXP);

InitEventProc(STATE_CALLING, RESPONSE_3XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Calng_RESPONSE_3_6XX);

InitEventProc(STATE_CALLING, RESPONSE_4XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Calng_RESPONSE_3_6XX);

InitEventProc(STATE_CALLING, RESPONSE_5XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Calng_RESPONSE_3_6XX);

InitEventProc(STATE_CALLING, RESPONSE_6XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Ev_Calng_RESPONSE_3_6XX);
```

```
InitEventProc(STATE_CALLING, TRANSPORT_ERR,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Calng_TRANSPORT_ERR);

 // state STATE_PROCEEDING message handlers
InitEventProc(STATE_PROCEEDING, RESPONSE_1XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Proc_RESPONSE_1XX);

InitEventProc(STATE_PROCEEDING, RESPONSE_2XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Proc_RESPONSE_2XX);

InitEventProc(STATE_PROCEEDING, RESPONSE_3XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Proc_RESPONSE_3_6XX);

InitEventProc(STATE_PROCEEDING, RESPONSE_4XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Proc_RESPONSE_3_6XX);

InitEventProc(STATE_PROCEEDING, RESPONSE_5XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Proc_RESPONSE_3_6XX);

InitEventProc(STATE_PROCEEDING, RESPONSE_6XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Proc_RESPONSE_3_6XX);

 // state STATE_COMPLETED message handlers
InitEventProc(STATE_COMPLETED, TIMER_D_EXPIRED,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Comptd_TIMER_D_EXP);

InitEventProc(STATE_COMPLETED, RESPONSE_3XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Comptd_RESPONSE_3_6XX);

InitEventProc(STATE_COMPLETED, RESPONSE_4XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Comptd_RESPONSE_3_6XX);

InitEventProc(STATE_COMPLETED, RESPONSE_5XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Comptd_RESPONSE_3_6XX);
InitEventProc(STATE_COMPLETED, RESPONSE_6XX_T,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Comptd_RESPONSE_3_6XX);

InitEventProc(STATE_COMPLETED, TRANSPORT_ERR,
 (PROC_FUN_PTR)&InviteClientTE::Evt_Comptd_TRANSPORT_ERR);

 // unexpected messages message handler
InitUnexpectedEventProc(STATE_INITIAL,
 (PROC_FUN_PTR)&InviteClientTE::Event_UNEXPECTED);

InitUnexpectedEventProc(STATE_CALLING,
 (PROC_FUN_PTR)&InviteClientTE::Event_UNEXPECTED);

InitUnexpectedEventProc(STATE_PROCEEDING,
 (PROC_FUN_PTR)&InviteClientTE::Event_UNEXPECTED);

InitUnexpectedEventProc(STATE_COMPLETED,
 (PROC_FUN_PTR)&InviteClientTE::Event_UNEXPECTED);
}

void InviteClientTE::Evt_Init_INVITE() {
 SendMessageToTPL();
 if (!IsTransportReliable()){
  TimerADuration = GetT1();
  setTimerCount(TIMER_A, TimerADuration);
  StartTimer(TIMER_A);
 }
 setTimerCount(TIMER_B, 64*GetT1());
 StartTimer(TIMER_B);
 MakeLocalCopyOfMsg();
```

```
 SetState(STATE_CALLING);
}

void InviteClientTE::Evt_Calng_TIMER_A_EXP(){
 TimerADuration = 2 * TimerADuration;
 setTimerCount(TIMER_A, TimerADuration);
 RestartTimer(TIMER_A);
 RetransmitInvite();
}

void InviteClientTE::Evt_Calng_RESPONSE_1XX(){
 uint16 val;
 StopTimer(TIMER_A);
 StopTimer(TIMER_B);
 SendMessageToTU();
 GetParamWord(INDEX_TLI_PARAM, val);
 SetIndexTLI(val);
 SetState(STATE_PROCEEDING);
}

void InviteClientTE::Evt_Calng_RESPONSE_2XX(){
 StopTimer(TIMER_A);
 StopTimer(TIMER_B);
 SendMessageToTU();
 SetState(STATE_INITIAL);
}

void InviteClientTE::Evt_Calng_TIMER_B_EXP(){
 StopTimer(TIMER_A);
 SendErrorMessageToTU();
 SetState(STATE_INITIAL);
}

void InviteClientTE::Evt_Calng_TRANSPORT_ERR(){
 StopTimer(TIMER_A);
 StopTimer(TIMER_B);
 SendErrorMessageToTU();
 SetState(STATE_INITIAL);
}

void InviteClientTE::Evt_Calng_RESPONSE_3_6XX(){
 uint16 val;
 StopTimer(TIMER_A);
 StopTimer(TIMER_B);
 SendMessageToTU();
 GetParamWord(INDEX_TLI_PARAM, val);
 SetIndexTLI(val);
 SendAckMessageToTPL();
 if (IsTransportReliable())
  setTimerCount(TIMER_D, ZERO_TIMER_VAL_APPROX);
 else
  setTimerCount(TIMER_D, 64*GetT1());//64T1
 StartTimer(TIMER_D);
 SetState(STATE_COMPLETED);
}

void InviteClientTE::Evt_Proc_RESPONSE_1XX(){
 SendMessageToTU();
}

void InviteClientTE::Evt_Proc_RESPONSE_2XX(){
 SendMessageToTU();
 SetState(STATE_INITIAL);
}
```

```
void InviteClientTE::Evt_Proc_RESPONSE_3_6XX(){
 SendMessageToTU();
 SendAckMessageToTPL();
 if (IsTransportReliable())
  setTimerCount(TIMER_D, ZERO_TIMER_VAL_APPROX);
 else
  setTimerCount(TIMER_D, 64*GetT1()); //64T1
 StartTimer(TIMER_D);
 SetState(STATE_COMPLETED);
}

void InviteClientTE::Evt_Comptd_TIMER_D_EXP(){
 SetState(STATE_INITIAL);
}

void InviteClientTE::Evt_Comptd_RESPONSE_3_6XX(){
 SendAckMessageToTPL();
}

void InviteClientTE::Evt_Comptd_TRANSPORT_ERR(){
 StopTimer(TIMER_D);
 SendErrorMessageToTU();
 SetState(STATE_INITIAL);
}

void InviteClientTE::Event_UNEXPECTED() {
}

void InviteClientTE::RetransmitInvite(){
 SendCopiedMessageToTPL();
}
```

The mandatory function *Initialize()* starts by setting the FSM initial state *STATE_INITIAL*. It then initializes the timers A, B, and D by calling the FSM Library function *InitTimerBlock()* (its parameters are the timer identification, the timer interval duration, and the identification of the associated message; see also Section 6.8.74). The function *Initialize()* finishes by setting the FSM state transition functions. These functions process various message types in different states, as follows:

- The function *Evt_Init_INVITE()* processes the message *INVITE* in the state *STATE_INITIAL*.
- The function *Evt_Calng_TIMER_A_EXP()* processes the message *TIMER_A_EXPIRED* in the state *STATE_CALLING*.
- The function *Evt_Calng_RESPONSE_1XX()* processes the message *RESPONSE_1XX_T* in the state *STATE_CALLING*.
- The function *Evt_Calng_ RESPONSE_2XX()* processes the message *RESPONSE_2XX_T* in the state *STATE_CALLING*.
- The function *Evt_Calng_TIMER_B_EXP()* processes the message *TIMER_B_EXPIRED* in the state *STATE_CALLING*.
- The function *Evt_Calng_RESPONSE_3_6XX()* processes the messages *RESPONSE_3XX_T*, *RESPONSE_4XX_T*, *RESPONSE_5XX_T*, and *RESPONSE_6XX_T* in the state *STATE_CALLING*.

- The function *Evt_Calng_TRANSPORT_ERR()* processes the message *TRANSPORT_ERR* in the state *STATE_CALLING*.
- The function *Evt_Proc_RESPONSE_1XX()* processes the message *RESPONSE_1XX_T* in the state *STATE_PROCEEDING*.
- The function *Evt_Proc_RESPONSE_2XX()* processes the message *RESPONSE_2XX_T* in the state *STATE_PROCEEDING*.
- The function *Evt_Proc_RESPONSE_3_6XX()* processes the messages *RESPONSE_3XX_T*, *RESPONSE_4XX_T*, *RESPONSE_5XX_T*, and *RESPONSE_6XX_T* in the state *STATE_PROCEEDING*.
- The function *Evt_Comptd_TIMER_D_EXP()* processes the message *TIMER_D_EXPIRED* in the state *STATE_COMPLETED*.
- The function *Evt_Comptd_RESPONSE_3_6XX()* processes the messages *RESPONSE_3XX_T*, *RESPONSE_4XX_T*, *RESPONSE_5XX_T*, and *RESPONSE_6XX_T* in the state *STATE_COMPLETED*.
- The function *Evt_Comptd_TRANSPORT_ERR()* processes the message *TRANSPORT_ERR* in the state *STATE_COMPLETED*.
- The function *Event_UNEXPECTED()* processes all unexpected messages in all states.

As we can see from the source code above, the state transition functions (message handlers) are short and easily readable because each program statement is easily traceable back to the original statechart and SDL diagrams. For example, consider the first state transition function *Evt_Init_INVITE()*. The original SDL specification of this state transition starts with the reception of the message *INVITE* (Figure 3.75). This step is provided by the class *FSMSystem*. The next step in the SDL diagram says: "Invite_T to TPL." This step is implemented with a single program statement, namely, the function call to the function *SendMessageToTPL()*.

The next step in the SDL diagram is the question, "Is transport reliable?" We implement it also with a single function call to the function *IsTransportReliable()*. We continue the SDL coding in this manner. If the transport is reliable, the initial value of the timer A is provided by calling the function *GetT1()*—a way to parameterize the software. Next, we set the timer A duration by calling the function *setTimerCount()*—this is the undocumented FSM Library function at the moment, to be included in the next official release—and start the timer A by calling the function *StartTimer()* (the parameter of this function is the timer identification; see also Section 6.8.138).

At the end of this function, we set the duration of the timer B and start it, make the local copy of the last sent message by calling the function *MakeLocalCopy()*—remember that it is needed for the possible retransmission—and set the new state by calling the function *SetState()* (its parameter is the state identification; see also Section 6.8.137).

Next, the state transmission function *EvtCalng_TIMER_A_EXP()* performs the reaction to the timer A expiration (see the corresponding SDL specification in Figure 3.75) with only four program statements. The first one doubles the timer A duration, the second sets this new duration, the third restarts the timer A by calling the FSM Library function *RestartTimer()* (see Section 6.8.87), and the fourth retransmits the message *INVITE* by calling the function *RetransmitInvite()*. Also, all the other state transition functions are made in this spirit of one-to-one mapping from the original SDL diagram. The reader is advised to study them as an additional exercise.

References

Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, 1998.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

5

Test and Verification

The test and verification phase is a phase of communication protocol engineering work that follows the implementation phase. The primary goal of this phase is to verify that the implementation in the higher-level programming language is correct. The implementation is correct if it meets its original requirements, which are modeled in the form of use cases (see Chapter 2).

The correctness of the implementation is checked with the test suite, which is typically designed in TTCN-3 (see Section 3.9). The test suite itself is implemented in a higher-level programming language, e.g., Java or C++. But how do we verify the correctness of the test suite implementation? The answer is that we do not check the correctness of the test suite independently. We always check the correctness of the implementation under the test and test suite simultaneously. Theoretically, a bug in a test suite can cover a bug in the implementation; we should be aware of this, but such cases seldom happen in practice.

Typical testing activities conducted in the communication protocol engineering test and verification phase are the following:

- Unit testing
- Integration testing
- Conformance testing
- Load testing
- In-field testing
- Formal verification
- Statistical usage testing

The first four types of activities (unit testing, conformance testing, load testing, and in-field testing) stem from traditional software engineering, whereas the last two (formal verification and statistical usage testing) originate from Cleanroom engineering. Today, communication protocol engineers tend to complement software engineering with Cleanroom engineering testing approaches, therefore we cover all the above listed activities in this chapter.

As its name suggests, unit testing is used for testing individual software units before their integration into the product. Typically, a software unit is a

single class written in a separate Java compilation unit or C++ module. This class most commonly implements a simple communication protocol or part of a more complex communication protocol. In the case of the FSM Library–based paradigm, such a unit would be a C++ module that defines the class derived from the class *FiniteStateMachine*.

Unit testing of communication protocols is relatively straightforward. Typically, we construct a set of test cases that check individual FSM state transitions, as well as more complex FSM transactions (series of FSM state transitions). We will use JUnit and CppUinit testing frameworks for unit testing of communication protocols in this book. Details of unit testing are given in Section 5.1 (Unit Testing) and Section 5.5.1 (Example 1).

The next phase is integration testing. The philosophy of integration testing starts from the fact that some of the units have successfully undergone unit testing and that they are available for further testing, whereas the rest of them are not. For the purpose of integration testing, we introduce replacements for the units that are not available, which are referred to as the imitators (or simulators).

There are two kinds of imitators, namely drivers and stubs. A driver is an active imitator that generates input messages for the real objects (units) under test. A stub is a passive imitator that accepts the output messages generated by the objects under test. Stubs can also send replays that are expected from the objects they are imitating. Of course, we can construct more complex imitators that act as both drivers and stubs. In this book, we will call the collaborations of real objects, *drivers,* and stubs simply *integration test collaborations*.

Generally, communication protocols are well suited for integration testing because families of communication protocols are hierarchically organized in layers with well-defined interfaces. The communication between individual protocols is based on messages, which are traditionally exchanged through the mailboxes (as in implementations based on the FSM Library). Simulating the environment of a real object under test in such a situation is easy. Drivers and stubs simply exchange messages with objects under test. Actually, they act on behalf of the units that will communicate with the units under test in the final product.

Normally, protocol stacks are implemented in the bottom-up fashion, starting from the lowest layer of the protocol stack and building the next layer on top of the previous one. Drivers and stubs in such an approach simulate only a part of the environment, the higher layer of the protocol stack in particular. An example of simple integration test collaboration is given in Section 5.5.2 (Example 2).

When all software units have undergone unit and integration testing, the final product is integrated and ready for acceptance testing, which comprises conformance testing (also referred to as compliance testing), load testing, and in-field testing. Preliminary acceptance testing can be organized solely by the production organization and conducted on its premises. However,

final acceptance testing is organized and conducted by the organization that has the legal authority to issue acceptance certificates.

As suggested by its name, the aim of conformance testing is to prove that the product (implementation) under test conforms to the original requirements. In the area of communication protocol engineering, these requirements would normally be standards issued by the IETF, ISO, ITU-T, ETSI, and similar organizations. The newer standards made by ITU-T and ETSI most frequently include the conformance test suite specification in TTCN-3.

Conformance testing is a kind of functional testing (also referred to as black box testing). The testers are not interested in the structure of the product and its internal behavior. They only ensure that the external behavior of the product meets the original specification. Typically, this behavior is specified with the set of scenarios described in TTCN-3. We will return to the subject of conformance testing in Section 5.2.

The load testing typically involves exposing the implementation under test to the conditions of the real exploitation. Conceptually, this means that the implementation under test must service the requests coming from more independent sources simultaneously. While conformance testing focuses on the correctness of services given to the minimal number of request sources, load testing checks the correctness of services driven by the requests coming from independent sources, preferably in an interleaved fashion.

Normally, load testing is conducted in a laboratory-simulated environment. Typically, we would construct, purchase, or lease the specialized equipment referred to as a load generator. A load generator is normally a programmable device that offers a selection of predefined scenarios and their parameters (such as number of request sources, duration of individual communication phases, and so on) as well as definitions of completely new scenarios.

The name *load generator* may be misleading because it suggests that the device generates only the requests—which it does—but it also receives the responses from the implementation under test and checks if it operates correctly. For example, after the connection is successfully established, it sends and receives test tones to check that the connection is really usable. During load testing, we primarily check declared traffic capabilities of the product. A typical requirement would be that the number of lost requests must not exceed the given limit after the given number of requests has been issued in accordance with the given request arrival distribution.

We also normally check the behavior of the implementation under test for both lower and higher rates of request arrivals. With an extremely low rate of requests, we want to check the sustainability of long-lasting connections, whereas with an extremely high rate, we want to make sure that the overload protection mechanisms are in place and that they function correctly. After successful load testing, the implementation under test is integrated into the

target network for in-field testing. In-field testing is essentially the experimental exploitation of the product for the given interval of time (e.g., three months).

The aim of in-field testing is to detect, locate, and eliminate bugs that are exposed by the real-world scenario (also referred to as a traffic case) that could not be simulated in the laboratory. During this last phase of acceptance testing, log files always prove to be extremely useful. Today, the log files can be collected over the Internet and analyzed remotely. Also, installing software upgrades can be done by uploading new software patches over the Internet.

Detecting bugs through the analysis of the log files can be augmented by adding program hooks for certain, really infrequent traffic cases. Defining state transition preconditions, postconditions, and invariants and checking them at run-time is also extremely useful for detecting bugs during in-field testing, and later during normal system exploitation. Although communication protocol maintenance is an integral part of communication protocol engineering, it is out of scope of this book (see directions for further reading in Section 5.6).

Traditional software engineering comprises a number of development phases, such as requirements, analysis, design, implementation, unit test, integration, integration test, verification, and maintenance. These phases can be cascaded in the case of the waterfall process model or revisited in the case of the spiral-incremental process model. The number of remaining bugs is the main software quality metric. Another important metric used in software engineering is test coverage (measured as the percentage of tested software paths, variable usages, and so on).

Cleanroom engineering, in contrast to traditional software engineering, is organized as a sequence of the following development activities:

- Formal model development.
- Formal verification of the formal model.
- Handing formal model to the implementation team, which implements it in a higher-level programming language.
- Operational profile modeling.
- Automatic test suite generation, which is based on the given operational profile model.
- Statistical usage testing and software reliability estimation: If at least one test case from the automatically generated test suite fails, the implementation under test is thrown away and the complete development cycle is repeated from the very beginning (starting with the formal model development).

The complete treatment of formal modeling and verification is out of the scope of this book (see directions for further reading in Section 5.6). As a means of introduction to the area of formal methods, formal modeling and

verification based on theorem proving and model checking is covered in Section 5.3, which is divided into Subsections 5.3.1 and 5.3.2. The paradigm described in the subsection 5.3.1 is based on the application of the theorem prover named THEO, whereas the paradigm described in the subsection 5.3.2 is based on the model checker PAT.

Operational profile modeling, automatic test suite generation, statistical usage testing, and software reliability estimation are described in Section 5.4. The paradigm described in that section is based on the application of the software tool, which is named generic test case generator (GTCG).

5.1 Unit Testing

The aim of unit testing is to check the correctness of an individual software unit (Java compilation unit or C/C++ module). A generally accepted belief, especially among proponents of agile methods such as extreme programming, is that unit testing should be conducted by the programmer who is implementing the target software unit, because it greatly improves programmer's productivity. In principle, unit tests should be written before, or at least during, the implementation of the target software unit.

Of course, the programmer must clearly distinguish between the roles of an implementer and a tester (the author of extreme programming, Kent Beck, uses the metaphor: "by changing hats" to explain this paradigm). The programmer, as unit tester, concentrates on the unit interface. By thinking about the interface and by writing unit tests, the programmer gets a clearer picture about the services that the target software unit must provide. The programmer should also try to make test cases that cover boundary conditions, as well as situations that would be potentially hard to manage for the target software unit.

The programmer, as the unit implementer, concentrates on the implementation of the original unit design. They should forget about unit tests and concentrate on mapping the design to code. This should be a straightforward task if a proper framework (such as the FSM Library) is provided.

Unit testing helps programmers produce software units of better quality in shorter time intervals and this has been proven in practice. First, by creating unit tests, the programmer becomes even more familiar with the implementation at hand. Second, the programmer gets immediate feedback. If there is a bug, it is easy to detect in the scope of a particular test case. If the test case passes, the programmer gets immediate satisfaction that they have done their job well.

Unit test cases should be executed frequently during the target unit's implementation. As time passes, new test cases are added and old cases are run again. Even if no new test cases are used, we should rerun all existing

unit tests every time we add new functionality. Testing that is conducted by running an unchanged test suite to check if the new software functionality has not affected existing functionalities is referred to as regression testing.

Regression testing is the key point of this paradigm. It enables a dramatic increase of productivity because it builds the programmer's confidence that everything is in good order and under control; therefore, the programmer can work more relaxed. Regression testing also encourages experimenting. In situations when alternative paths may be used in the course of implementation, the programmer may try out a way that seems most appropriate. If one or more test cases fail in the regression testing that is subsequently conducted, the programmer may decide to reset to the starting point by retrieving the previous version from the installed version of the control system database.

Unit testing (including regression testing) definitively has a positive impact on a programmer's psychology. It is estimated to be the key factor for increases in the programmer's productivity. The next question is "to what extent should we go with the unit testing?" The answer is not easy. Certainly, any amount of unit testing is better than none. Alternately, an attempt at exhaustive unit testing might be counterproductive.

The right choice is somewhere between these two extremes. We do not need to test trivial things, such as class function members that set or get the value of a certain private field member. Rather, we should concentrate on the boundary conditions and parts of code where it becomes more complex. Although generally unpopular among professionals, copy–paste practice may be tolerated for generating a set of similar test cases.

Three principal preconditions exist for a successful unit testing practice:

- A proper unit testing framework must be provided.
- Test cases should not involve any human intervention.
- The implementation under test must not be changed.

A proper unit testing framework must provide three main functions:

- Test case registration: This function enables registering new test cases within the given test suite hierarchy. On each level of the hierarchy, a set of individual test cases may be found, as well as other hierarchically subordinated test suites (very similar to the file system structure).
- Test case execution: This function provides automatic execution of all test cases defined within the given test suite hierarchy. It must not require more than a single push button to be started. Otherwise, the framework is simply not usable.
- Test case reporting: This function must provide a general report on the outcome of the execution of all test cases, as well as individual reports for all test cases that failed or caused errors.

The second precondition is that test suite execution should not involve any human intervention. This is the essential precondition to make unit testing completely automatic. If we want to eliminate human intervention, we must secure two conditions: First, the input data required by a test case must be defined as symbolic constants in its source code or in other external files. Second, the results of the test case must be automatically checked by a test case itself. The unit testing framework must provide adequate functions for this purpose.

A typical function for checking test case results is the function *assert(condition)*, where *condition* is a Boolean expression that evaluates to either the value *true* or *false*. The test case continues (*pass*) in the former case and breaks (*fail*) in the latter case. If the test case execution successfully reaches the end of the test case, it is considered successful (qualified with the verdict *pass*). Otherwise, it is considered unsuccessful (qualified with the verdict *fail*). If the test case execution breaks because of some error (most typically, an exception such as "divide by zero"), it is qualified with the verdict *error*.

Another typical function for checking test case results is the function *assertEquals(p1,p2)*. This function call is semantically equivalent to the function call *assert(p1==p2)*. This means that if the parameters *p1* and *p2* are equal (of course, they must be comparable), the test case execution continues; otherwise, it breaks. Typically, one of the parameters is a constant and another is a program variable.

Although these two functions are semantically equivalent, the function *assertEquals()* is advantageous when it comes to test case reporting. If the function *assert()* breaks the test case execution, the unit testing framework reports only that the condition evaluated to the value *false*, which is not a very informative report. Alternately, if the function *assertEquals()* breaks, the framework provides the report "expected *C* but was *V*," where *C* is the value of the constant (e.g., *p1*) and *V* is the real value of the variable (e.g., *p2*).

We can further improve the readability of the test case execution reports by using the optional text string parameter of the function *assertEquals()*. Generally, the function call format for this function is *assertEquals(text, condition)*, where *text* is the text string that explains the meaning of this assertion point in more detail. The string *text* is used as a prefix of the test report shown above. For example, if the value of the variable *ch* should be 'A' but it turns out to be 'B' instead, the function call *assertEquals("Check ch:," 'A', ch)* would produce the report, "Check ch: expected 'A' but was 'B.'"

Besides the functions *assert()* and *assertEquals()*, unit testing framework typically provides two additional functions for writing test cases: *setUp()* and *tearDown()*. The former sets up the test fixture whereas the latter destroys it. A test fixture is a set of objects that act as samples for testing. Normally, the test fixture comprises the instance of the unit under test (e.g., the instance of the class that is derived from the class *FiniteStateMachine*) and also other supplementary objects, which are required for effective unit testing.

Typically, the unit testing framework offers the base class for writing test cases, which provides the functions *assert()*, *assertEquals()*, *setUp()*, and *tearDown()*. The programmer normally derives his tester class from this base class, fills in *setUp()* and *tearDown()* functions, and starts writing individual test cases. Each function member of the tester class—whose name follows the given naming convention—is a single test case.

Remember that concrete *setUp()* and *tearDown()* implementations are shared by all test cases defined within a single tester class. Actually, these two functions are implemented as null (empty) methods on test cases. The execution of each test case starts with the call to the function *setUp()*, proceeds with a call to the user-defined function that implements a single test case, and ends with the call to the function *tearDown()*. Normally, we put the test case initialization and cleanup code in the functions *setUp()* and *tearDown()*, respectively.

The third unit testing postulate is that the unit under test must not be touched at all. We are only allowed to write new classes that are derived from the base class, which is provided by the unit testing framework. Changing the source code of the unit under test for the purpose of its testing is strictly forbidden, even by adding a simple print statement to the standard output file. Because of that, the only proper way to do the unit testing is to drive the unit under test with various messages, capture its responses, and check the correctness of the unit's external behavior.

This kind of controlled execution of the implementation under test is referred to as the test harness. The key request is that it must be fully automatic. The programmer should provide the mechanisms that support the test harness while he plays the role of the implementer (what we refer to as the *design for testability*). Otherwise, providing a test harness can be a very hard task. For example, consider a simple program that reads its input from the keyboard and writes its output to the monitor by using the operating system services, which cannot be replaced. Because we are not allowed to change the source code of the implementation under test, providing a test harness in this case is hardly achievable.

An example of the unit testing framework is JUnit, an open-source testing framework for unit testing Java programs that was originally developed by Erich Gamma and Kent Beck. Based on this framework, the open-source community came up with CppUnit, a semantically equivalent testing framework for unit testing C++ modules. These frameworks are very simple but powerful enough to enable industrial-strength unit testing of individual software units. Because JUnit and CppUnit are semantically equivalent, we will treat them as two implementations of the same framework.

The framework comprises the interface *Test* and two fundamental classes, the classes *TestSuite* and *TestCase*, as in Figure 5.1. As shown in the figure, the test suite (an instance of the class *TestSuite*) can contain an arbitrary number of test cases (instances of the class *TestCase*), as well as an arbitrary number of other hierarchically subordinated test suites. This arrangement allows

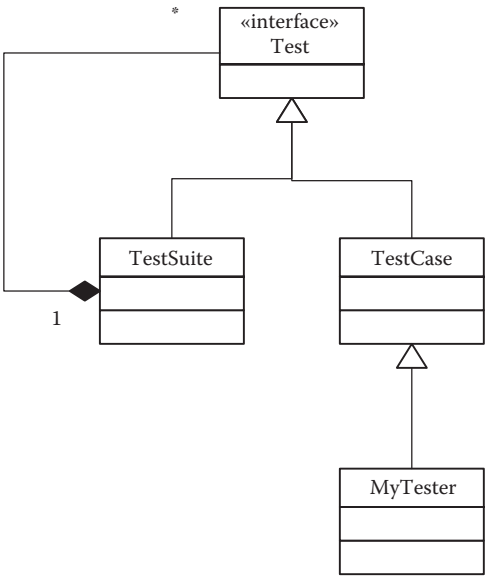


FIGURE 5.1
Structure of the JUnit testing framework.

programmers (playing the role of unit testers) to organize test cases into a hierarchy of test suites to their convenience.

Any concrete tester class (such as the class *MyTester* in Figure 5.1) must be derived from the base class *TestCase*, which, among others, provides the four fundamental functions described above, namely, *setUp()*, *tearDown()*, *assert()*, and *assertEquals()*. By convention, an individual test case is written as the function member of the tester class, whose name starts with the word "test," for example, *test1*, *test2*, and so on.

Next, we illustrate JUnit's usability on a concrete example. In the example that follows, we demonstrate the unit testing paradigm for the case where the implementation under test is counter by modulo 2. The particular implementation we are interested in is the one based on the State design pattern. This implementation was presented in Section 4.3.

As already mentioned in Section 4.3, the function *processMsg()*, which processes FSM input (message), prints its results by calling the function member *println()* of the class *MyIO*, rather than by calling the standard I/O function *System.out.println()*. This is a good example of how we can provide support for the test harness in our design and implementation. Here is the source code of the class *MyIO*:

```
package automata4;
import java.util.*;

public class MyIO {
private static String lastOutput;
```

```
 public static String getLastOutput() { return lastOutput; }
 public static void println(String s) {
 lastOutput = s;
 System.out.println(s);
 }
}
```

The field member *lastOutput* is used to store the last output generated by the FSM. The function *getLastOutput()* returns this last output generated by the FSM to its caller. It is used by the test case function to retrieve the last FSM output to compare it with the expected output (also referred to as the "golden output"). The function *println()* is simple enough—it just stores the output of the FSM and prints it by calling the standard function *System.out.println()*.

Although we do not need it in this example, we can generally use an analogous approach for capturing the FSM inputs also. Instead of calling the standard function *System.in.read()* directly, we can construct and call the function member *read()* of the class *MyIO*. This function would, in its own turn, read the input by calling the standard input functions and store that input into the corresponding field member of the class *MyIO* (e.g., *lastInput*). The last FSM input would be available through the function member *getLastInput()*.

After providing test harness support, we continue with the definition of the tester class, which is named *Automata4Tester* in this example. The source code of this class is as follows:

```
/*
* Automata4 tester
*
*/

package automata4;
import junit.framework.*;

public class Automata4Tester extends TestCase {
 protected Automata4 a4;
 public Automata4Tester(String name) {
 super(name);
  }

  protected void setUp() {
   // setup code
   a4 = new Automata4();
  }

  protected void tearDown() {
   // cleanup code
  }

  // test case 1
  public void test1() {
   a4.processMsg('0');
   assertEquals(MyIO.getLastOutput(),"Output 0");
   a4.processMsg('0');
   assertTrue(MyIO.getLastOutput() == "Output 0");
  }
```

```
  // test case 2
  public void test2() {
   for(int i=0;i<100;i++) {
   a4.processMsg('0');
   assertEquals(MyIO.getLastOutput(),"Output 0");
   }
  }

  // test case 3
  public void test3() {
   a4.processMsg('0');
   assertEquals(MyIO.getLastOutput(),"Output 0");
   a4.processMsg('1');
   assertEquals(MyIO.getLastOutput(),"Output 1");
   a4.processMsg('0');
   assertEquals(MyIO.getLastOutput(),"Output 1");
   a4.processMsg('1');
   assertEquals(MyIO.getLastOutput(),"Output 2");
   a4.processMsg('0');
   assertEquals(MyIO.getLastOutput(),"Output 2");
   a4.processMsg('1');
   assertEquals(MyIO.getLastOutput(),"Output 0");
  }
  // test case 4
  public void test4() {
   a4.processMsg('1');
   assertEquals(MyIO.getLastOutput(),"Output 1");
   a4.processMsg('1');
   assertEquals(MyIO.getLastOutput(),"Output 2");
   a4.processMsg('1');
   assertEquals(MyIO.getLastOutput(),"Output 0");
  }

  // test case 5
  public void test5() {
   for(int i=0;i<1000;i++) {
     test3();
     test4();
    }
  }
  public static TestSuite suite() {
   return new TestSuite(Automata4Tester.class);
    }

  public static void main(String[] args) {
   junit.textui.TestRunner.run(suite());
  }
}
```

The tester class *Automata4Tester* is derived from the class *TestCase*. Its field member *a4* is an instance of the implementation under test, namely, the class *Automata4*. The constructor of the class *Automata4* simply calls the constructor of its super class (the class *TestCase*) and passes its input parameter (*String name*).

The function *setUp()* creates an instance of implementation under test by instantiating the class *Automata4*, and storing its instance into the field member *a4*. The function *tearDown()* is empty in this example because the Java garbage collector takes care of unused objects. The garbage collector

destroys the object that is stored in the field member *a4* at the end of the test case.

The function *test1()* is the first test case defined within the tester class *Automata4Tester*. Basically, it tests the FSM state transition from the state *S0* to the state *S0*, which is driven by the input value *0*. It does the same operation twice. It supplies input *0* to the implementation under test (stored in the field member *a4*) each time by calling its function *processMsg()* and passing it the parameter, '0'.

Assuming that the implementation under test was in its initial state and that it reacted correctly to the given input, its last output should be the text, "Output 0". The test case function *test1()* checks that assumption by calling the function *assertEquals()*. The first real parameter of that function call is the value of the last output, which is returned by the function member *getLast Output()* of the class *MyIO*, whereas the second parameter is the expected string, "Output 0".

Second, the test case function *test1()* again supplies input *0* to the implementation under test (stored in the field member *a4*) by calling its function *processMsg()* and passing the parameter '0' to it. Assuming that the implementation under test has reacted properly in the first place, it would be in the initial state at the time the second call to the function *processMsg()* happens. Driven with the input '0', it should again produce the output string "Output 0". The test case function *test1()* checks this assumption again, only this time it does so by calling the function *assert()*. The real parameter of this function call is the condition *MyIO.getLastOutput() == "Output 0"*.

The function *test2()* is the second test case defined within the tester class *Automata4Tester*. This test case is slightly more complex than the previous one. The previous test case checks if the implementation under test reacts correctly when it is driven twice with the same input value *'0'* in the same current state (*S0*). We did this on purpose—first, to demonstrate the usage of both *assert()* and *assertEquals()*, and second, the implementation under test may not always react correctly if it is driven with a certain input value in the given state, at least not in theory.

This practice may seem paranoid but, in reality, various types of time- and FSM evolution-dependent bugs are hidden at the beginning and become evident only later during the FSM evolution. Returning to the problem at hand, we ask ourselves: Will this FSM react correctly many times, for example, 100 times? With JUnit at our disposal, we can easily construct a test case that resolves such dilemmas.

This is exactly what the test case function *test2()* does. It does so by executing the body of the *for* loop 100 times. Inside the body of the loop, it drives the implementation under test with input value '0' by calling its function *processMsg()*. After each of these calls, it checks if the last output was the string "Output 0" by calling the function *assertEquals()*.

The function *test3()* is the third test case defined within the tester class *Automata4Tester*. This is a typical FSM-related test case, characterized with complete coverage of the FSM state transition graph. The flow of the state transitions checked by this test case is the following:

- From S0 to S0, driven with the input 0 (expected output 0)
- From S0 to S1, driven with the input 1 (expected output 1)
- From S1 to S1, driven with the input 0 (expected output 1)
- From S1 to S2, driven with the input 1 (expected output 2)
- From S2 to S2, driven with the input 0 (expected output 2)
- From S2 to S0, driven with the input 1 (expected output 0)

The function *test4()* is the fourth test case defined within the tester class *Automata4Tester*. This is another typical FSM-related test case, characterized by its progressive nature. The counter is always driven with the input "1" so that its content is incremented every time. This test case does not provide the full state transition graph coverage, but it is valid, and we can think of many partial graph coverage test cases. The flow of the state transitions checked by this test case is as follows:

- From S0 to S1, driven with the input 1 (expected output 1)
- From S1 to S2, driven with the input 1 (expected output 2)
- From S2 to S0, driven with the input 1 (expected output 0)

The function *test5()* is the fifth, and the last, test case defined within the tester class *Automata4Tester*. It is a fairly simple, yet rather intensive, test case that is based on the combination of the previous two test cases. The test case function *test5()* repeats the body of the *for* loop 1,000 times. Inside the body of the loop, it just calls the functions *test3()* and *test4()* in succession.

The function *suite()* returns the test suite, which it creates by calling the constructor of the class *TestSuite*. The real parameter of this function call is the name of the implementation under test class file (*Automata4Tester.class*). The constructor of the class *TestSuite* finds all the functions whose names start with the word "test" defined within the class *Automata4Tester* and automatically adds them to the test suite it creates.

The function *main()* runs the test suite defined by the previous function *suite()*. It does that by calling the function *run()* of the class *TestRunner*, which is an integral part of the JUnit testing framework. The real parameter of this function call is the test suite that is created by the function *suite()*. This test suite contains all test cases defined within the class *Automata4Tester*.

In the case of more complex implementations, we may decide to create more tester classes rather than define all test cases within a single tester

class, such as the class *Automata4Tester*. In such a situation, we would need to create a hierarchy of test suites and an overall tester class that would automatically run all test cases in all test suites. The source code of such a tester class is the following:

```
/*
* Tester
*
*/

package automata4;
import junit.framework.*;

/*
* TestSuite that runs all test suites
*
*/

public class AllTests {
  public static void main (String[] args) {
    junit.textui.TestRunner.run(suite());
  }
  public static TestSuite suite() {
    TestSuite suite = new TestSuite("All Tests");
    suite.addTest(Automata4Tester.suite());
    // add other test suites here
    return suite;
  }
}
```

The class *AllTests* comprises two function members, namely, the functions *suite()* and *main()*. The former function creates and returns the test suite that is in the root of the test suite hierarchy. This means that it contains all other hierarchically subordinated test suites. The latter function executes the root test suite, i.e., it executes all test suites that were added to it.

The function *suite()* creates the root test suite simply by calling the constructor of the class *TestSuite*. The real parameter of this function call is the name of that test suite (the string "All Tests"). It then adds the test suite that contains the test cases defined within the tester class *Automata4Tester* to the root test suite. It does this by calling the function member *addTests()* of the root test suite object *suite*. Generally, in the case when we have multiple tester classes, we would repeat the call to the function *addTests()* for each tester class.

The function *main()* runs the test suite defined by the previous function *suite()*. It does this by calling the function member *run()* of the class *TestRunner*. The real parameter of this function call is the test suite created by the function member *suite()* of the class *AllTests*. This test suite contains a single, hierarchically subordinated test suite, which in turn contains all test cases defined within the class *Automata4Tester*.

We start the automatic execution of all test cases defined within the class *Automata4Tester* by running the file *Automata4Tester.class*. Similarly, we start the automatic execution of all test cases defined within all tester classes (in this simple example, we have just one of them: the class *Automata4Tester*) by

running the file *AllTests.class*. In both cases, we should get the same result. Each test case function will print its own outputs to the standard output file. At the end, the test runner will print out the final report, which should look like this:

```
Time: 1,783
OK (5 tests)
Press any key to continue...
```

The number 1783 corresponds to the number of seconds that were needed to execute all test cases, whereas the number 5 in parenthesis corresponds to the total number of test cases that were executed.

5.2 Conformance Testing

As already mentioned at the beginning of this chapter, conformance testing is the first step of acceptance testing (followed by load testing and in-field testing). The aim of conformance testing is to check the functional correctness of external behavior of the implementation under test without checking its inner workings. Essentially, conformance testing is functional testing that is based on the "black box" approach.

The main goal of conformance testing is to separately check the correctness of each individual function of the implementation under test (IUT). The sample test case for a simple SIP softphone (IUT) is: "Initiate session setup. Check if IUT sends the message INVITE to the outbound proxy server (imitated by the testing framework). Make the testing framework replay with the message 404 (not found). Check if IUT replays with the message ACK" (see sequence diagram in Figure 5.2). We are intentionally making test cases

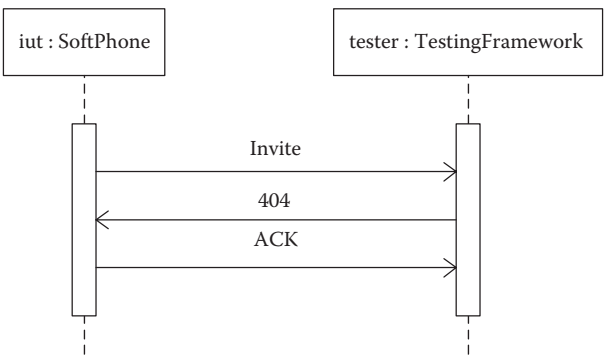


FIGURE 5.2
Example of the conformance testing test case.

as simple as possible so we can easily interpret their outcomes. Of course, some of the test cases are inevitably complex and we cannot do anything about this, but we should never make them more complex than they need to be.

More precisely, we do not try to check more functions simultaneously by interleaving the corresponding scenarios. For example, consider the SIP proxy server as the implementation under test. In the case of conformance testing, we are only interested if it can support a single session establishment at a time. Normally, we would not be interested in checking if it can support multiple session establishments simultaneously. Actually, that is exactly the purpose of load testing.

When it comes to specifying official conformance test suites for real-world protocols (like SIP), this is a really serious business conducted by the international standardization institutions, such as IEEE, ISO, IETF, ITU-T, ETSI, and others. The results are rather voluminous specifications that most frequently use TTCN language. The most recent version of TTCN at the time of this writing is the TTCN-3 (see Section 3.9), which enables both tabular and program formats of specifications.

For a better understanding of the scope of conformance testing, consider the documents currently available from ETSI (you can download them from the Internet; see http://www.etsi.org) that are related to conformance testing of SIP (IETF RFC 3261). These documents are the following:

- Conformance test specification for SIP, Part 1: Protocol implementation conformance statement proforma (ETSI TS 102 027-1)
- Conformance test specification for SIP, Part 2: Test suite structure and test purposes (ETSI TS 102 027-2)
- Conformance test specification for SIP, Part 3: Abstract test suite and partial protocol implementation of extra information for testing (ETSI TS 102 027-3)

The first document is the proforma to be completed by the vendor of the implementation to claim implementation capabilities. The guidance for completing the proforma is given in Section 5. This document is used both during static conformance review and during the test suite parameterization phase of conformance testing.

The second document describes the test suite structure and the purposes of individual test cases. This document was used as the test plan before the test suite was written in the TTCN-3 language. Now it is used as the reference document for understanding the abstract test suite, which is given in the third document.

The third document specifies the abstract test suite to be used for SIP conformance testing. Actually, it is composed of two files, the archive (ZIP file) that contains SIP test suite in TTCN-3 program format, and the SIP test suite

overview file (PDF file). The SIP test suite in TTCN-3 program format can be executed using a commercially available TTCN-3 tool.

The SIP conformance test suite specification by ETSI (the three documents listed above) considers four types of implementations under test. The implementations are as follows (see IETF RFC 3261 for their definitions):

- User agent that behaves as client or server
- Registrar
- Proxy server (both outbound and simple proxy server)
- Redirect server

The present version of the specification considers the following three types of sessions:

- Sessions that are established using a proxy server
- Sessions that are established directly (without proxy)
- Sessions that are established using the redirect server

The way the SIP conformance test suite is structured is a good example of typical conformance test suite structuring. All test cases are classified into the following four main groups (which correspond to the main SIP functionalities):

- Registration
- Call control
- Querying for capabilities
- Messaging

The test cases in the main groups are further classified according to the role that should be checked. The roles for the main group *registration* are the *registrant* and the *registrar*. The roles for the main group *call control* are *originating endpoint*, *terminating endpoint*, *proxy*, and *redirect server*. The roles for the main group *querying for capabilities* are *originating endpoint*, *terminating endpoint*, and *proxy*. The roles for the main group *messaging* are *registrant*, *registrar*, *originating endpoint*, *terminating endpoint*, *proxy*, and *redirect server*.

Some of the role subgroups are further divided into functional subgroups. For example, the role subgroup *originating endpoint* of the main group *call control* is divided into three functional subgroups, namely, *call establishment*, *call release*, and *session modification*. Finally, functional subgroups of test cases can be divided into three test groups: *valid behavior* (V), *invalid behavior* (I), and *inopportune behavior* (O).

Notice that official conformance testing can be conducted only by authorized organizations (national certification centers, telecom operators, and so

on) that use special tools that themselves were certified for such usage. These tools are professional equipment, most frequently referred to as testers, e.g., a SIP tester. A tester typically comprises the framework that supports test suite administration, execution (most frequently based on interpretation), and associated reporting. Such a framework is referred to as the testing framework.

The testers may be rather sophisticated. Most of them support most of— if not all—the state-of-the-art protocols. Alternately, almost unique testers are also used that support ultramodern protocols that have not become part of the mainstream protocols. Both of these types of testers can be rather expensive. Most frequently, competent and efficient operating of protocol testers requires special training.

Because of that, most of the small- and even middle-scale organizations involved in protocol development cannot afford purchasing testers and employing full-time employees (confusingly enough, also called testers) for the purpose of conformance testing. Rather, they rent the equipment or the person who can operate it for the purpose of the unofficial and preliminary conformance testing at the client location. The goals of this preliminary conformance testing are to reduce the overall cost and to minimize the risk of failing the official conformance testing.

Some organizations use open source test suites to reduce the cost of the preliminary conformance testing. An example of such a test suite is the SIP Forum Basic UA Test Suite created by Nils Ohlmeier, freely available on the Internet at https://github.com/nils-ohlmeier/sipsak (in accordance with the GPL license). This test suite is comprised of the following two parts:

- SIP Forum Testing Framework (SFTF)
- Basic UA tests

SFTF provides regular functions of test suite administration (e.g., adding new test cases, simply referred to as "the tests"), test suite execution control (executing all tests, selected groups of tests, or individual tests), and test suite execution reporting (both by printouts in the interactive window and in the log files, with five possible levels of logging details). The testing framework contains the logic required to execute the test, parse incoming messages, and create replies.

The second part (listed above) is simply a subdirectory that contains all basic user agent tests (i.e., test cases). The tests and SFTF itself are written in Python. The goal of these tests is not to provide complete conformance testing of SIP implementations, as the ETSI specification does. Rather, the goal is to check the well-known SIP interoperability problems, which frequently occur in immature SIP User Agent (UA) implementations, such as the simple SIP softphone.

Additionally, these tests can discover the implementation under test behavior that conforms to the original SIP specification but is considered

a suboptimal implementation solution. Such cases are reported as *warnings* (W). The developer should consider revising the implementation in the case of warnings to make it more robust.

Many tests in this test suite are adopted from the IETF's SIP torture tests Internet draft (available on the Internet under the name *draft-ietf-sipping-torture-tests-02*). The rest of the tests are the contributions from the SIP Forum members. Original IETF SIP torture tests focus on areas that have caused problems in the past or have particularly unfavorable characteristics if handled improperly. Some of them test only the parser and others test both the parser and the application above it. Some use valid and some use invalid SIP messages to check target functionality.

The SIP Forum tests are classified into the following eight test groups: protocol tortures (26 tests), authentication (4 tests), registration (1 test), dialog and transaction processing (19 tests), DNS (2 tests), NAT capabilities (2 tests), services (2 tests), and warnings about obsolete features (5 tests). All tests are defined in one spreadsheet (XLS file). The test attributes (spreadsheet columns) are the following: number, title, tested device, expected behavior, typical failures, notes, call flow, source (the corresponding section in RFC 3261), and comment.

For example, the test number 201 entitled "A Short Tortuous Request" tests the SIP user agent server behavior. The expected behavior is, "Server considers the request valid and generates a proper response". The call flow is illustrated with the sequence diagram shown in Figure 5.3.

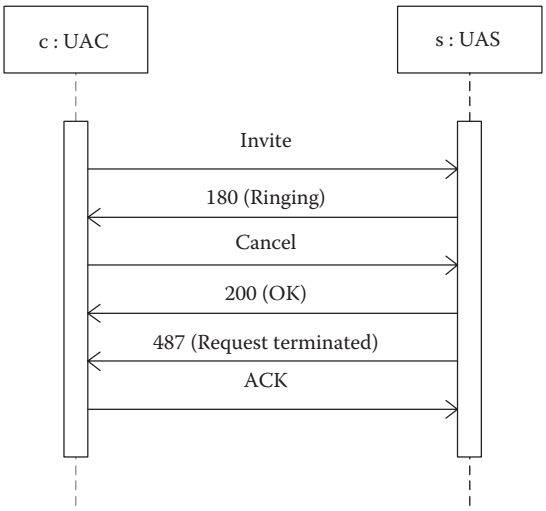


FIGURE 5.3
Example of the SIP protocol torture test.

5.3 Formal Verification

5.3.1 Formal Verification Based on Theorem Proving

This section covers the formal verification of communication protocols based on automated theorem proving. The reader will learn how to use automated theorem proving for formal verification of both communication protocol specification and its implementation. Normally, the communication protocol is modeled as the finite state machine. Basic knowledge of predicate calculus (first-order logic) is assumed for easy and complete understanding of this section.

The outline of this section is the following:

- Axiomatic specification of finite state machines
- Theoretic specification of test cases
- Formal verification of the specification
- Directions for generating test cases
- Formal verification of the implementation
- Software development process based on the formal verification
- A realistic example

The axiomatic specification of the finite state machine is the model of the FSM in the predicate calculus. This model is the set of well-formulated formulas. The first well-formulated formula in the model is optional and it defines the initial state of the FSM. Its general format is the following:

```
State(INITIAL).
```

State is a predicate and *INITIAL* is the name (label) of the FSM initial state. The names *State* and *INITIAL* are non-interpretative user-defined names (like names of the user-defined functions and constants in the higher-level programming languages). For brevity, in this section we use the name *S* instead of *State* and we label finite state machine states with numbers (*0*, *1*, *2*…) rather than with symbolic names.

The fact that this first well-formulated formula is optional requires a short comment. In most of the formal FSM descriptions, such as UML activity diagrams and statecharts, the specification of the FSM initial state is mandatory. Here, it is not. If we always want to examine the FSM evolution, beginning from the same state, we will define it as the FSM initial state in the FSM axiomatic specification. Alternately, sometimes it is possible and preferable to examine the FSM evolution beginning from different FSM states. In that case, we do not define the FSM initial state in the FSM axiomatic specification; instead we define it on the left-hand side of the concluding well-formulated formula.

The rest of the well-formulated formulas in the FSM axiomatic specification are obligatory. Each of the mandatory well-formulated formulas models

a single FSM state transition (also referred to as a FSM branch). The format of the well-formulated formula that models the time invariant FSM state transition from the state *X* to the state *Y* triggered with the input *T* and generating the output *R* is as follows:

$$\{State(X)\&Input(T)\} => \{State(Y)\&Output(R)\}$$

State, *Input*, and *Output* are predicates. *X*, *Y*, *T*, and *R* are constants that label the source FSM state, the destination FSM state, the particular FSM input, and the particular FSM output, respectively. Most frequently, we use abbreviated names *I* and *S* instead of *Input* and *Output*, respectively. In the case that the state transition generates more, say *N*, output signals (messages), the corresponding well-formulated formula has the following format:

$$\{State(X)\&Input(T)\} => \{State(Y)\&Output(R\_1)\& Output(R\_2)\&\ldots\&Output(R\_N)\}$$

where *R_1*, *R_2*…*R_N* are the labels of particular output signals.

Next, we introduce the concept of control predicates. As their name suggests, the control predicates are used to control the FSM activity. A global control predicate is used to enable or disable the complete FSM activity. Usually we name it *A(N_I)*, where *A* stands for *Automata* and *N_I* labels the particular FSM.

Besides the global control predicate, state transition control predicates also exist, one for each FSM state transition. A state transition control predicate enables or disables the associated state transition. We typically name it *T(M_I)*, where *T* stands for *Transition* and *M_I* labels the particular FSM state transition. The state transition well-formulated formula that includes control predicates has the following format:

$$\{Automata(I)\&Transition(J)\&State(X)\&Input(T)\}=>\{State(Y)\&Output(R)\}$$

I is the label of the particular FSM and *J* is the label of the particular state transition modeled with this formula. If we include both *Automata(I)* and *Transition(J)*, the state transition is enabled. If we skip *Automata(I)*, the FSM (i.e., all its state transitions) are disabled. If we skip *Transition(J)*, this individual state transition is disabled. This concludes the presentation of the axiomatic specification of a single FSM.

A theoretical test case for a single FSM is the theorem about the particular FSM evolution path, which states that for a given series of inputs (I_1, I_2…I_n), FSM performs a series of state transitions (S_1, S_2…S_n), which will produce a series of particular output values (O_1, O_2…O_n). The corresponding well-formulated formula has the following format:

$$\{Automata(N)\&Transition(M)\&Input(I_1)\&\ldots\&Input(I_n)\} => \{Output(O_1)\&...\&Output(O_n)\&State(S_1)\&\ldots\&State(S_n)\}$$

Most frequently, we only want to check that FSM produces the expected series of outputs and that at the end it reaches the expected final state S_n. The corresponding theorem has a very similar, but simpler format:

$$\{Automata(N)\&Transition(M)\&Input(I_1)\&\ldots\&Input(I_n)\} \Rightarrow \{Output(O_1)\&...\&Output(O_n)\&State(S_n)\}$$

Before proceeding to modeling the groups of communicating FSMs, let us look at a simple example. The following shows the axiomatic specification of the counter by modulo 2 (see the statechart diagram in Figure 5.4) and a sample theorem about its expected behavior. The FSM axiomatic specification is as follows:

S(0)
$\{A(0)\&T(0)\&S(0)\&I(0)\} \Rightarrow \{S(0)\&O(0)\}$
$\{A(0)\&T(1)\&S(0)\&I(1)\} \Rightarrow \{S(1)\&O(1)\}$
$\{A(0)\&T(2)\&S(1)\&I(0)\} \Rightarrow \{S(1)\&O(1)\}$
$\{A(0)\&T(3)\&S(1)\&I(1)\} \Rightarrow \{S(2)\&O(2)\}$
$\{A(0)\&T(4)\&S(2)\&I(0)\} \Rightarrow \{S(2)\&O(2)\}$
$\{A(0)\&T(5)\&S(2)\&I(1)\} \Rightarrow \{S(0)\&O(0)\}$

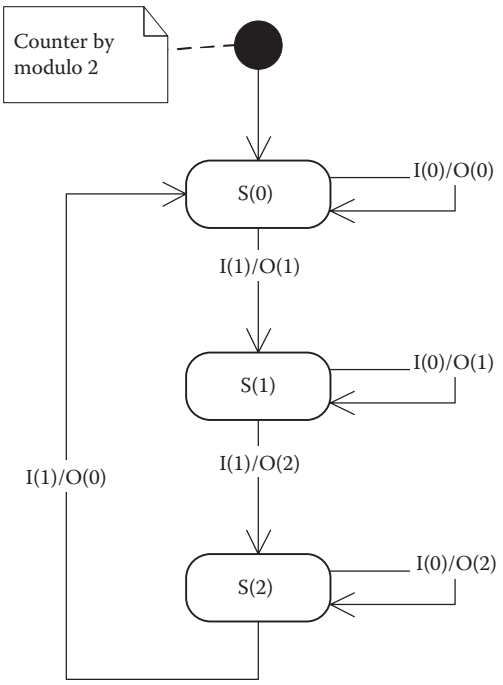


FIGURE 5.4
Counter by modulo two statechart.

The first well-formulated formula defines the state S(0) as the FSM initial state. Next, six well-formulated formulas define six FSM state transitions—from the state *S(0)* to *S(0)*, from *S(0)* to *S(1)*, from *S(1)* to *S(1)*, from *S(1)* to *S(2)*, from *S(2)* to *S(2)*, and from *S(2)* to *S(0)*, respectively. *A(0)* is the global control predicate. *T(0)*, *T(1)…T(5)* are the individual state transition control predicates. The sample theorem is as follows:

$$\{A(0)\&T(0)\&I(0)\&T(1)\&I(1)\} => \{O(0)\&O(1)\&S(1)\}$$

It may be interpreted as follows: The FSM is globally enabled by including the general control predicate *A(0)* on the left-hand side of the concluding well-defined formula. The first FSM state transition is enabled by including the state transition predicate *T(0)*. The FSM is stimulated with the input *I(0)*, which should result in the output *O(0)*. The second FSM state transition is enabled by including the state transition control predicate *T(1)*. The FSM is stimulated with the input *I(1)*, and the FSM should generate *O(1)* at its output. Finally, the FSM should reach the state *S(1)*.

We can prove this theorem with the automated theorem prover THEO developed by Monty Newborn (2001). To do that, we must write the theorem in a text file, compile it using the program Compile (*cc.exe*), and prove it by running the program THEO (*teo.exe*). The final result looks like this:

```
Predicates: S A T I O
Functions: 0 1 . 2 3 4 5 :
EQ:
ESAF:
ESAP:
 0 <BC: 19 NC: 6 AC: 3 U: 0>
 1 {T0 N1 R1 F0 C9 H0 h0 U11} *
.Proof Found!
```

Of course, realistic FSMs never operate in isolation. Rather, they normally operate in groups of cooperating finite state machines. For example, according to ITU-T, the system consists of functional blocks interconnected with communication channels (see Section 3.7, SDL). Each functional block comprises finite state machines (processes) interconnected with signaling paths (routes). A communication channel may comprise one or more signaling paths. Finite state machines communicate by exchanging signals (events, messages) over signaling paths.

We can use such a kind of traditional system decomposition for our convenience, but it is not required. In the opposite extreme, we can have a chaotic system in which each FSM talks to all other FSMs (like stations in wireless networks). We can even connect more FSMs in signaling networks with all kinds of topologies, such as start, bus, or a network that connects an arbitrary number of FSMs. The means to model all these abstractions in the first-order logic are predicates and their compositions.

To start, we can introduce the notation *Signal(SIG_N)* that represents the act of signaling the particular signal, where *Signal* is a predicate and

SIG_N is the label of a particular signal. We then can introduce the notation *SignalOverPath(SIG_N,PATH_M)* that represents the act of signaling the particular signal over the particular signaling path, and so on. The well-formulated formulas that model state transitions do not change much. For example, the state transition from the state *X* to the state *Y* is triggered with the signal *P* and generates the signal Q, and looks like this:

{*State(X)*&*Signal(P)*} => {*State(Y)*&*Signal(Q)*}

In the formula above, *Signal(P)* is received and *Signal(Q)* is sent out of any signaling path, channel, or network. In the case where the former signal is transferred over path *M* and the latter signal is sent over the path *N*, the formula would look like this:

{*State(X)*&*SignalOverPath(P,M)*} => {*State(Y)*&*SignalOverPath(Q,N)*}

After introducing the concept of signaling between finite state machines in a group of cooperating FSMs, we can proceed to the axiomatic specification of the group of FSMs. As shown above, each FSM in a group is specified with a set of well-formulated formulas (one optional for the initial state and one mandatory for each individual state transition). Consequently, the specification of a group of FSMs is the union of sets of well-formulated formulas for individual FSMs that constitute that group.

The theoretical test case for the group of FSMs is just a generalization of the theoretical test case for the individual FSM. The left-hand side of the corresponding well-formulated formula consists of control predicates, if any, and staring signals whereas the right-hand side of the formula lists the resulting signals and final states of individual FSMs. The format of the typical theorem about the evolution of the group of FSMs is as follows (assume the system with two FSMs):

{*Signal(A)*} => {*Signal(B)*&*Signal(C)*&*Signal(D)*&*State(X)*&*State(Y)*}

In the sample theorem above, *Signal(A)* triggers the evolution of the system. As the result of the evolution, the system generates three signals: *Signal(B)*, *Signal(C)*, and *Signal(D)*. At the end of the evolution, the FSMs reach their final states, namely, *State(X)* and *State(Y)*.

We now illustrate the concepts introduced above by the means of a simple example. Consider a simple system with three FSMs (see their statechart diagrams in Figure 5.5). The first FSM waits for the signal *E(0)* in its state *S(0)*. After receiving that signal, it sends the signal *E(10)* and goes to the state *S(1)*, where it waits for the signal *E(1)*. Once it receives the signal *E(1)*, it sends the signal *E(20)* and goes to the state *S(2)*. The second and the third FSMs are very much alike. The former waits for the signal *E(10)* and, after receiving that signal, it sends the signal *E(11)*. The latter waits for *E(20)* and sends *E(21)*.

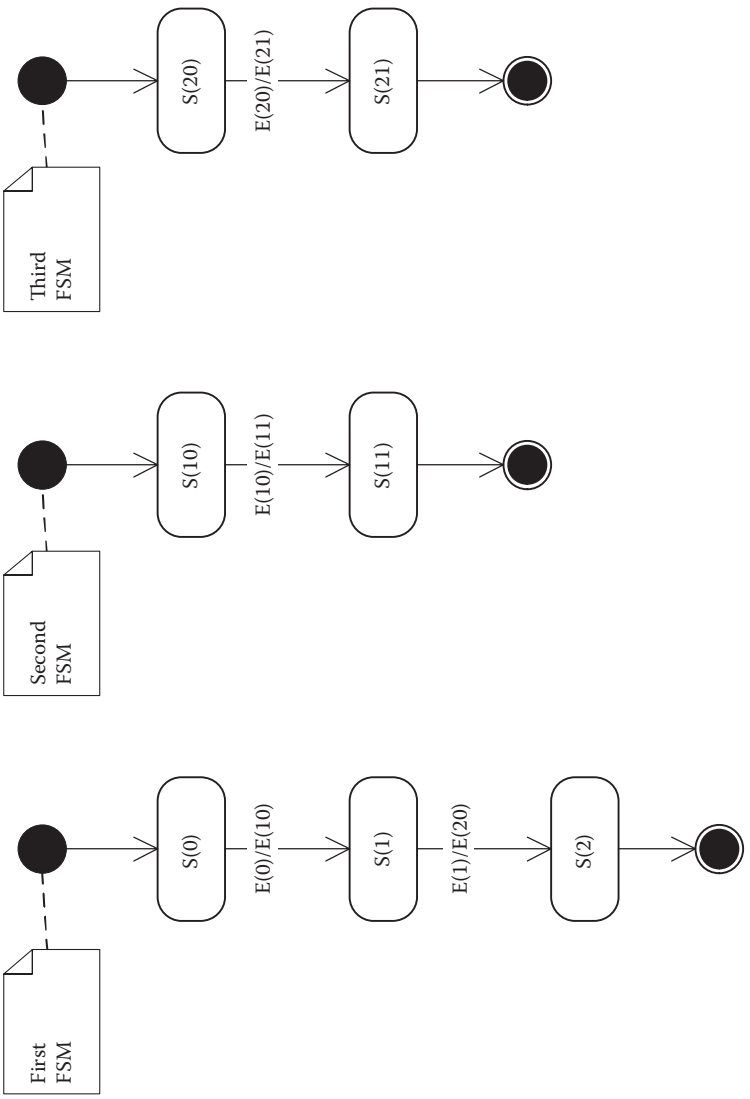


FIGURE 5.5
Statecharts of three communicating FSMs.

Next, we construct the theorem about the expected behavior of this simple system. This theorem says that if we supply signals *E(0)* and *E(1)* to this system, the first FSM will start evolving and will generate the signals *E(10)* and *E(20)*. These two signals will trigger the second and the third FSMs, which will, in turn, generate signals *E(11)* and *E(21)*, respectively. Finally, these FSMs will reach final states *S(2)*, *S(11)*, and *S(21)*, respectively.

The axiomatic specification of this simple system and the theorem explained above are specified in the following sequence of well-formulated formulas:

```
; Simple system with 3 FSMs
; Axiomatic spec. of the first FSM
S(0).
{S(0)&E(0)} => {S(1)&E(10)}.
{S(1)&E(1)} => {S(2)&E(20)}.

; Axiomatic spec. of the second FSM
S(10).
{S(10)&E(10)} => {S(11)&E(11)}.

; Axiomatic spec. of the third FSM
S(20).
{S(20)&E(20)} => {S(21)&E(21)}.

; Theorem
conclusion
{E(0)&E(1)} => {S(2)&S(11)&S(21)&E(10)&E(20)&E(11)&E(21)}.
```

To automatically prove this theorem, we run Compile and THEO once again. The final result looks like this:

```
Predicates: S E
Functions: 0 1 10 2 20 11 21 : .
EQ:
ESAF:
ESAP:
 0 <BC: 14 NC: 3 AC: 3 U: 0>
 1 {T0 N1 R1 F0 C1 H0 h0 U14} *
.Proof Found!
```

Next, we introduce the concept of a theoretical log file. As already mentioned, a theoretical test case is a theorem about an FSM's expected behavior. It defines starting (input) signals on its left-hand side and a series of expected output signals and traversed FSM states (including the final ones that we are most interested in) on its right-hand side. We refer to the right-hand side of the theoretical test case as the theoretical log file.

A strong similarity exists between the theoretical and the real log files. The real log file is the result of the system execution in real time. It represents a particular path of the system evolution. The theoretical log file is the result of the virtual (speculative) system execution. It shows the expected outcomes, such as generated signals and traversed states (including the final states).

However, one principal difference between the two of them is that the logs in the real log file usually have a time stamp. The value of the time stamp

is usually unique (with the exception of the logs in multiprocessor systems). Alternately, logs in the theoretical log files are individual predicates that correspond to signals and states, and they do not have any time stamp at all.

Furthermore, we can write logs in the theoretical log file in any order, because the operator "&" is a commutative one. The easiest way to think about it is that the theoretical test case is true forever. Hence, it really does not matter in which order we name the logs. Another way to think about it is that all of them have happened at the same moment of time. Therefore, all logs have the same "time stamp," which may be omitted because it does not provide any meaningful information, and then again the order of logs does not matter.

Actually, when we look at the FSM axiomatic specification, and the theoretical test case more closely, we notice that no explicit notion of time exists at all. The only notion of time present there is an implicit one, and it is made through control predicates. Although the absence of an explicit notion of time may seem confusing and disadvantageous, it is the main source of the power of proving theorems.

To understand why, imagine that we made a system that reacts in certain ways when it receives two different messages, but we are not sure what will happen if these two signals arrive at exactly the same time. If the probability of this event is very low, it can take a long period of time before the event happens and we face a system failure. With the theorem-proving approach, we check such situations immediately. Imagine the enormous amounts of test time that are saved this way.

Another powerful characteristic of this approach is that each theoretical test case actually represents a family of test cases. For example, let us return to the counter by modulo 2. Consider the theorem:

$$\{A(0)\&T(0)\&I(0)\} \Rightarrow \{O(0)\&S(0)\}$$

Because in first-order logic, $I(0) \Leftrightarrow I(0)\&I(0)$, we can rewrite the theorem as follows:

$$\{A(0)\&T(0)\&I(0)\&I(0)\&I(0)\&I(0)\&I(0)\} \Rightarrow \{O(0)\&S(0)\}$$

We may interpret this theorem as follows: If we apply the same signal *I(0)* many times (even up to infinity), we will always get the signal *O(0)* at the FSM output and it will remain in the state *S(0)*. Therefore, by proving individual theoretical test cases, most frequently we are actually checking the families of test cases. This concludes the presentation of axiomatic specification and theoretical test cases related to FSMs.

Now let us see how we can use this in communication protocol engineering. We start with the formal verification of the specification. The concept is rather simple, although it can prove to be difficult to realize in practice. Ideally, two independent teams must be present (or at least a person who is

"changing hats"), namely, the design and testing teams. The former writes the axiomatic specification of the family of communication protocols that is modeled as a group of FSMs. The latter writes and proves the theoretical test cases.

If a theoretical test case fails (the proof of the theorem cannot be found), at least one error is generated in either axiomatic specification or in the theorem. It may be the case that two or even more errors occur in both of them. Most frequently, the errors are trivial oversights made by theorem writers because they are not so familiar with the system at hand. If not, the errors are typically caused by rather nontrivial oversights in the system design.

Finding these errors is not a trivial task at all. Typically, we would try to shorten the theorem or the axiomatic specification and see what happens. Of course, with an automated theorem prover, such as THEO, at our disposal, this is much easier than doing it by hand. Control predicates may help, also—with them, we can sequence the events to our convenience. The need for them is typically a clue that we have synchronization problems.

We can also use an automated theorem prover for automatic test case generation. To do that, we assume that axiomatic specification of the system is errorless. We start by selecting one of the possible input signals on the left-hand side of the theorem. We then check various output signals at the right-hand side of the theorem by trying to prove the theorem. If the proof is found, our assumption was correct and we keep that signal at the right-hand side. If not, we continue by checking other signals.

Of course, some input signals can just cause internal state transitions and no signals at the output of the system. The right-hand side will remain empty in that case. By continuing this process, we can generate theoretical test cases of arbitrary length:

$$\{I(A)\&I(B)\&I(C)\} \Rightarrow \{O(X)\&O(Y)\&O(Z)\}$$

Similarly, we can make guesses about transient or final states of the system, for example:

$$\{I(A)\&I(B)\&I(C)\} \Rightarrow \{O(X)\&O(Y)\&O(Z)\&S(P)\&S(Q)\}$$

The real benefit of such automatically generated test cases is that they can be translated into executable test cases and used for automatic testing of the system implementation. Generating test cases in the previously described fashion is not very efficient, and neither it is well coordinated. We can generate test cases more cleverly by respecting the structure of the FSM axiomatic specification rather than viewing it as a black box. Actually, the FSM axiomatic specification introduced in this section is yet another means of modeling the FSM state transition graph.

Generating test cases by traversing the FSM state transition graph is possible with the goal to achieve its complete coverage. Three possible types of

FSM state transition coverage exist, namely, node, branch (arc), and path coverage. That the path coverage cannot be achieved if the graph is cyclic is well known. Alternately, branch coverage subsumes node coverage and, because of that, seems to be the best selection.

Sometimes we may have the opposite problem. The test suite (a set of test cases) may already be available, such as the SIP conformance test suite available from ETSI in TTCN-3 language (see Section 5.3). In such a situation, we can use a tool to translate TTCN-3 test cases into theorems, and then we can use the automated theorem prover to formally verify conformance of the system axiomatic specification with the standard.

Yet another application of the automated theorem prover is the formal verification of the system implementation. To do this, we assume that a conformance test suite is already available and use the reverse engineering tool to extract the axiomatic specification of the system from the implementation source code and, optionally, from log files if some are available. The reverse engineering tool normally relies on conventions that govern the structure of the source code and log files.

For example, the reverse engineering tool for the FSM Library-based implementations relies on the specification of the FSM Library API (see Section 6.8). This tool simply searches the source code for specific library functions and their real parameters to retrieve the well-formulated formulas that constitute system axiomatic specification. More precisely, the tool extracts the elements of the left-hand side of the state transition well-formulated formula by searching for library functions *InitEventProc()* and *InitUnexpectedEventProc()*.

The real parameters of the function *InitEventProc()* are the source state, the triggering signal (event, message), and the state transition function. The first two parameters (state and signal) are exactly the elements of the left-hand side of the corresponding well-formulated formula. The real parameters of the function *InitUnexpectedEventProc()* are the source state and the state transition function. The state is the first element of the left-hand side of the well-formulated formula. The second element is any signal that is not valid for the given state.

The reverse engineering tool proceeds by examining an individual state transition function. It creates one well-formulated formula (they all have the same left-hand side) for each state transition function execution path. For example, a state transition function with a simple sequence of statements yields a single formula, whereas a state transition function that has a switch with three cases yields three formulas.

The right-hand side of the state transition well-formulated formula is constructed by the analysis of the state transition function. The tool first searches for the functions *PrepareNewMessage()* and *SendMessage()* to extract symbolic names of the signals that are generated by that execution path of the state transition function. It then searches for the function *SetState()*, whose real parameter is the name of the destination state. If this function is not found, the tool assumes that the FSM state should not be changed and copies the state name from the left-hand side to the right-hand side of the formula.

This procedure is repeated for all state transition functions. Finally, the tool provides complete axiomatic specification of the system in ASCII format, which is readable by the automated theorem prover. We then use already available test cases to formally verify the system implementation source code.

Although most frequently we assume that the tools and other components we use are bug-free (in this particular case, these tools are the reverse engineering tool, compiler, linker, loader, and operating system), sometimes they are not. No matter how low the probability of such a failure is, it can happen and when it does, it compromises the formal verification of the source code. In such a case, we can use the reverse engineering tool that extracts the axiomatic system specification from log files. The example of the particular log file that was created by the FSM Library-based implementation is given in Section 5.5.1. Principally, the axiomatic specification that is provided from the log file is usually incomplete (except when it contains traces of all possible system execution paths), but even as such, it is sufficient to locate and eliminate the problem at hand.

When it comes to the application of formal verification methods, software development processes can be classified into three different categories. The Cleanroom engineering is a typical representative of the first category. It uses formal verification methods to formally verify the system design. The second category uses formal methods to formally verify the system implementation, whereas the third uses it to formally verify both the system design and implementation.

We will end this section with a more realistic example—the axiomatic specification of the FSM that implements both ITU-T Q.71 FE1 and FE5 call control functional entities (see Figure 3.38, Section 3.7.1) and a sample theoretical test case. The former functional entity models the functionality of the calling party (also referred to as subscriber A) whereas the latter models the functionality of the called party (also referred to as subscriber B). The following is the axiomatic specification of the FSM, named FE1FE5 (ITU-T Q.71 FE1 and FE5 merged together):

```
;
;  FE1FE5 definition
;
;  Initial state definition:
S(FE1FE5_ON_HOOK).

{S(FE1FE5_ON_HOOK)&E(r3_DisconnectReqInd)} =>
{S(FE1FE5_ON_HOOK)&E(r3_DisconnectRespConf)}.

{S(FE1FE5_ON_HOOK)&E(r3_SetupReqInd)} =>
{S(FE1FE5_WAIT_OFF_HOOK)&E(r3_ReportReqInd)}.

{S(FE1FE5_ACTIV)&E(r3_SetupReqInd)} =>
{S(FE1FE5_ACTIV)&E(r3_DisconnectReqInd)}.

{S(FE1FE5_ACTIV)&E(r3_DisconnectReqInd)} =>
{S(FE1FE5_WAIT_ON_HOOK)&E(r3_DisconnectRespConf)}.
```

```
{S(FE1FE5_ACTIV)&E(User_ON_HOOK)} =>
{S(FE1FE5_ON_HOOK?)&E(r3_DisconnectReqInd)}.

{S(FE1FE5_WAIT_ON_HOOK)&E(User_ON_HOOK)} =>
{S(FE1FE5_ON_HOOK)}.

{S(FE1FE5_WAIT_ON_HOOK)&E(r3_DisconnectReqInd)} =>
{S(FE1FE5_WAIT_ON_HOOK)&E(r3_DisconnectRespConf)}.

{S(FE1FE5_WAIT_ON_HOOK)&E(r3_SetupReqInd)} =>
{S(FE1FE5_WAIT_ON_HOOK)&E(r3_DisconnectReqInd)}.

{S(FE1FE5_WAIT_OFF_HOOK)&E(User_OFF_HOOK)} =>
{S(FE1FE5_ACTIV)&E(r3_SetupRespConf)}.

{S(FE1FE5_WAIT_OFF_HOOK)&E(r3_DisconnectReqInd)} =>
{S(FE1FE5_ON_HOOK)&E(r3_DisconnectRespConf)}.

{S(FE1FE5_WAIT_OFF_HOOK)&E(r3_SetupReqInd)} =>
{S(FE1FE5_WAIT_OFF_HOOK)&E(r3_DisconnectReqInd)}.

conclusion
; {S(FE1FE5_ON_HOOK)&E(User_OFF_HOOK)} =>
; {S(FE1FE5_UNKNOWN_FE2)&E(r1_SetupReqInd)}.

; {S(FE1FE5_UNKNOWN_FE2)&E(User_ON_HOOK)} =>
; {S(FE1FE5_DISCONNECTING_FE2)}.

{S(FE1FE5_ON_HOOK)&E(User_OFF_HOOK)&E(User_ON_HOOK)} =>
{S(FE1FE5_DISCONNECTING_FE2)&E(r1_SetupReqInd)}.
```

Actually, this file contains three theorems (starting after the keyword *conclusion*). The first two are commented out (the semicolon character ";" at the beginning of the line means that the line is a comment) leaving only the third open as a subject to prove by the automated theorem prover. The first commented theorem claims that if the FSM FE1FE5 is stimulated with the input signal *User_OFF_HOOK* in its initial state *FE1FE5_ON_HOOK*, it will generate the output signal *r1_SetupReqInd* and move to the state *FE1FE5_UNKNOWN_FE2*. The second commented theorem claims that if the FSM FE1FE5 is further stimulated with the signal *User_ON_HOOK* in the state *FE1FE5_UNKNOWN_FE2*, it will just move to the state *FE1FE5_DISCONNECTING_FE2*.

Finally, the third theorem—which is actually the subject of automated theorem proving—is a simple composition of the previous two theorems. It states that if the FSM FE1FE5 is stimulated by the sequence of the input signals *User_OFF_HOOK* and *User_ON_HOOK* in its initial state *FE1FE5_ON_HOOK*, it will generate the output signal *r1_SetupReqInd* and finish in the state *FE1FE5_DISCONNECTING_FE2*. To automatically prove this theorem, we run Compile and THEO once again. The final result looks like this:

```
Predicates: S E
Functions: FE1FE5_ON_HOOK User_OFF_HOOK r1_SetupReqInd User_ON_HOOK
FE1FE5_DISCONNECTING_FE2 . r1_DisconnectRespConf FE1FE5_UNKNOWN_FE2
r1_DisconnectReqInd User_DIGIT r1_ProceedingReqInd
FE1FE5_WAIT_FOR_DIGITS r1_ADDL_AddrReqInd r3_DisconnectReqInd
FE1FE5_WAIT_ON_HOOK r1_SetupRespConf FE1FE5_ACTIV r1_ReportReqInd
```

```
r3_DisconnectRespConf r3_SetupReqInd FE1FE5_WAIT_OFF_HOOK
r3_ReportReqInd FE1FE5_ON_HOOK? r3_SetupRespConf :
EQ:
ESAF:
ESAP:
0 <BC: 56 NC: 4 AC: 4 U: 0>
1 {T1 N1 R1 F0 C49 H1 h0 U8} *
.Proof Found!
```

5.3.2 Formal Verification Based on Communicating Sequential Processes

This section covers formal verification of communication protocols based on the process algebra named Communicating Sequential Processes (CSP) and aided by the toolkit named Process Analysis Toolkit (PAT). PAT supports a rich modeling language named CSP#, which is essentially the CSP extended with elements of the programming language C#. PAT also supports the First-Order Logic (FOL) and Linear Temporal Logic (LTL) formulas. Actually, PAT is a powerful toolkit comprised of many modules, including the module CSP#, the module Real-Time Systems (RTS), the module Probability CSP (PCSP), the module Probability RTS (PRTS), the module Labeled Transition Systems (LTS), the module Timed Automata (TA), the module NesC (targeting sensor networks), the module Orc (targeting Service Oriented Architecture), the module Stateflow (MDL), the module Security, the module Web Services (WS), and the module UML to PAT (for translating UML state machines to CSP#). We focus on the module CSP# in this book, because it is the most commonly used module.

In this section, the reader will learn from examples how to model protocols in CSP# and how to formally verify them by checking their desired properties, which are normally specified in the form of the corresponding FOL and/or LTL formulas. We will start with some more simple, classical examples (such as alternating bit protocol and two-phase commit protocol), we will continue with various leader election protocols (in complete graphs, in rings, and in rooted trees), and we will end with an example of a real-world communication protocol for providing telecomm services (such as basic call establishment and release, unconditional call forwarding, etc.).

The outline of this section is the following:

- Brief overview of CSP in Section 5.3.2.1
- Brief overview of PAT and CSP# in Section 5.3.2.2
- Examples of formal verification based on CSP and PAT in Section 5.3.2.3

5.3.2.1 Brief Overview of CSP

Process algebra is a formal method that uses an algebraic approach to study the communications of concurrent systems. Three well-known process

algebras are Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), and Algebra of Communicating Processes (ACP). This section serves as a brief introduction to CSP (Hoare, 1985), which was initially proposed by C.A.R. Hoare in 1978, and since then it has been developed into one of the most mature formal methods that are based on process algebras. CSP is specialized in modeling the interaction between concurrent systems using mathematical theories. Due to its powerful expressiveness, CSP is widely used in many different fields, such as real-time systems, web services, security, etc. CSP processes are composed of primitive processes and actions, which are connected by operators.

Here are the most important notions related to CSP processes:

- $\alpha P = \alpha(P)$ is the alphabet of the process P, i.e., the set of actions that P can engage in.
- αc is the set of messages that are communicable on channel c.
- $a \rightarrow P$ means that the process first performs the action a and then behaves as the process P. We read it as a then P.
- $(a \rightarrow P) \mid (b \rightarrow Q)$ means the choice between $(a \rightarrow P)$ and $(b \rightarrow Q)$, where b is the second action and Q is the second process.
- $(x : A \rightarrow P(x))$ means the choice of x from A then $P(x)$.
- $\mu\ X : A \bullet F(X)$ means the process X with the alphabet A such that $X = F(X)$.
- $P\ /\ s$ means P after engaging in events of trace s.
- $P \mid\mid Q$ means P in parallel with Q (i.e., the parallel execution of P and Q).
- $P\ [|X|]\ Q$ means that P and Q perform concurrent events on a set of channels X.
- $l : P$ means P with the name l.
- $L : P$ means P with names from the set L.
- $P \sqcap Q$ means the nondeterministic choice between P and Q. We read it as P or Q.
- $P \Box Q$ means the deterministic choice between P and Q. We read it as P choice Q.
- $P \setminus C$ means P without the elements of the set C.
- $P \mid\mid\mid Q$ means the interleaving of P and Q. We read it P interleaves Q.
- $P >> Q$ means P is chained to Q.
- $P\ //\ Q$ means P is subordinate to Q.
- $P\ ;\ Q$ means that P is successfully followed by Q. We read it P followed by Q.

- $P \lhd b \rhd Q$ means P if b (is true), else Q.
- $*P$ means repeat P (more precisely, P repeats an arbitrary number of times).
- $b * P$ means while b (is true) repeat P.
- $x := e$ means x becomes (the value of) e.
- $b!e$ means on (the channel) b output (the value of) e.
- $b?x$ means from (the channel) b input to x.
- $l!e?x$ means the call of the shared subroutine named l with the value parameter e and the results to x.
- P sat S means that the process P satisfies the specification S.
- *tr* is an arbitrary trace of the specified process, e.g., ⟨x, y⟩ where x and y are the elements of the alphabet of the specified process.
- *ref* is an arbitrary refusal of the specified process. The *refusal* of the process is the set of actions (events) in which the process cannot engage.
- $x^{\surd}$ means the final value of x produced by the specified process.
- $var(P)$ is the set of variables assignable by the process P.
- $acc(P)$ is the set of variables accessible by the process P.
- *Skip* is a process which does nothing but terminates successfully.
- *Stop* is a process which is in the state of deadlock and does nothing.

The most important notions related to CSP special events are the following:

- $\surd$ means success (successful termination of the specified process).
- $l.a$ means participation in event a by a process named l.
- $c.v$ means communication of the value v on the channel c.
- $l.c$ is the channel c of a process named l.
- $l.c.v$ means communication of the message v on the channel $l.c$.
- *acquire* means acquisition of a resource.
- *release* means release of a resource.

Formal system verification in CSP is based on the trace model of a process, which is a set of traces, where each trace represents a sequence of events that the process may perform. The most important notions related to process traces are as follows:

- ⟨⟩ is the empty trace.
- ⟨a⟩ is the trace containing only a (the singleton sequence).
- ⟨a, b, c⟩ is the trace with three symbols, a then b, then c.

- $\frown$ is the trace catenation operator in this book. For example, $\langle a, b, c\rangle = \langle a, b\rangle \frown \langle\rangle \frown \langle c\rangle$.
- s^n is the trace s repeated n times. For example, $\langle a, b\rangle^2 = \langle a, b, a, b\rangle$.
- $s \uparrow A$ means s restricted to A (in this book $\uparrow$ is used as a restriction operator). For example, $\langle b, c, d, a\rangle \uparrow \langle a, c\rangle = \langle c, a\rangle$.
- $s \leq t$ means s is a prefix of t. For example, $\langle a, b\rangle \leq \langle a, b, c\rangle$.
- $s \leq^n t$ means s is like t with up to n symbols removed. For example, $\langle a, b\rangle \leq^2 \langle a, b, c, d\rangle$.
- s in t means trace s is in the trace t (i.e., s is the subtrace of the trace t). For example, $\langle b, c\rangle$ in $\langle a, b, c, d\rangle$.
- $\#s$ means the length of trace s. For example, $\#\langle b, c, b, a\rangle = 4$.
- $s \downarrow b$ means the count of b in s. For example, $\langle b, c, b, a\rangle \downarrow b = 2$.
- $s \downarrow c$ means the communications on the channel c recorded in s. For example, $\langle c.1, a.4, c.3, d.1\rangle \downarrow c = \langle 1, 3\rangle$.
- $s\ ;\ t$ means the trace s successfully followed by the trace t. For example, $(s \frown \langle\surd\rangle)\ ;\ t = s \frown t$.
- A^* means the set of sequences with elements in A, or more formally $A^* = \{s \mid s \uparrow A = s\}$.
- s_0 means the head of s. For example, $\langle a, b, c\rangle_0 = a$.
- s' means the tail of s. For example, $\langle a, b, c\rangle' = \langle b, c\rangle$
- $s[i]$ means the ith element of s. For example, $\langle a, b, c\rangle[1] = b$.
- $f^*(s)$ means apply f on each element of s; we read it as f star of s. For example, $square^*(\langle 1, 5, 3\rangle) = \langle 1, 25, 9\rangle$.

The syntax of CSP core language is defined as follows:

$$P, Q ::= Skip \mid Stop \mid a \rightarrow P \mid P\ ;\ Q \mid c?x \rightarrow P \mid$$
$$c!x \rightarrow P \mid P \mid\mid Q \mid P\ [|X|]\ Q \mid P \lhd b \rhd Q$$

At the end of this section, let's have a look in some of the evergreen examples of CSP processes from Hoare (1985).

Example 1: The process *COPY*, which immediately copies every message it has input from the channel named *left* by outputting it to the channel named *right*.

$$\alpha left(COPY) = \alpha right(COPY)$$
$$COPY = \mu\ X\ \bullet\ (\ left?x \rightarrow right!x \rightarrow X)$$

The process *COPY* satisfies the following specification:

$$COPY \text{ sat } right \leq^1 left$$

Example 2: The process *DOUBLE* is like process COPY, except that every input number is doubled before it is output.

$\alpha left(DOUBLE) = \alpha right(DOUBLE) = N$
$DOUBLE = \mu X \bullet (left?x \rightarrow right!(x + x) \rightarrow X)$

The process *DOUBLE* satisfies the following specification:

$DOUBLE\ sat\ right \leq^1 double^*(left)$

5.3.2.2 Brief Overview of PAT and CSP#

PAT is an extensible and modularized framework for automatic system analysis based on CSP, which is freely available for noncommercial research at http://sav.sutd.edu.sg/PAT/. This self-contained framework supports modeling, simulating, and verifying concurrent real-time systems including communication protocols. PAT supports various model checking techniques targeting different properties, such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking, and probabilistic model checking. Moreover, the PAT development team implemented advanced optimization techniques, including partial order reduction, symmetry reduction, process counter abstraction, and parallel model checking, in order to achieve good performance from the user point of view.

The main PAT facilities are as follows:

- Multidocument and multilanguage editor for creating models
- Simulator for visual and interactive simulation of system behaviors, including random simulation, step-by-step simulation, complete state graph generation, trace playback, and counterexample visualization
- Verifiers for deadlock-free analysis, reachability analysis, state/event LTL checking (with and without fairness assumptions), and refinement checking
- Documentation and many examples (we focus on some of them in the next section)

The PAT framework has been developed by J. Sun, Y. Liu, J.S. Dong, and their colleagues at the National University of Singapore since 2007 (Sun 2009). The first time that PAT was successfully demonstrated internationally was at the 30th International Conference on Software Engineering in 2008. Over the last decade, many other researches worldwide have been using PAT to model and verify various systems, ranging from recently proposed distributed algorithms and security protocols to real-world systems like multilifts and pacemakers. However, in this book, we focus on communication protocols.

We continue with a brief overview of CSP#. CSP# (pronounced "CSP sharp") is a super-language for CSP, which combines high-level operators (mostly coming from CSP) such as conditional and nondeterministic choices, interrupt, parallel composition, interleaving, hiding, and asynchronous message passing, with low-level C# programming language constructs such as variables, arrays, and control flow statements like if–then–else and while and for loops. CSP# supports both general models of communication among processes, namely *shared memory* and *message passing*. The former model of communication is supported by the means of global variables, whereas the second model is supported by the channels for asynchronous message passing or by the CSP multiparty barrier synchronization. The main CSP# design principle is to keep the original CSP as a core sublanguage and additionally to provide access to data states and executable data operators from C#.

The CSP# language constructs may be divided into the following four groups:

- The core subset of CSP operators, including event-prefixing, internal and external choices, alphabetized lock-step synchronization, conditional branching, recursion, etc.
- The language constructs that are regarded as a syntactic sugar to CSP, including global variables and asynchronous channels: Although, the original CSP supports modeling of shared variables and asynchronous channels as processes, the dedicated language constructs offer better usability and improved verification efficiency.
- The set of event annotations: Since CSP supports only the notion of safety, the event annotations provide additional means for modeling fairness using event-based compositional language.
- The language constructs for stating assertions that may be automatically verified using PAT built-in verifiers.

The language syntax structures are classified as follows:

- Global definitions
- Process definitions
- Assertions

5.3.2.2.1 CSP# Global Definitions

CSP# global definitions include:

- Model names
- Global constants
- Global variables and arrays

- Asynchronous channels
- Macros

Model names are given using a declaration **//@@** *model_name* **@@**.

Constants are specified using the C macro directive **#define** or the key word **enum**. A constant value may be either integer or Boolean Examples:

```
#define N 10; // N == 10
enum {zero, one, two}; // zero == 0, one == 1, two == 2
```

Variables and arrays are specified using the keyword **var**. Since CSP# is a weekly typed language, no typing information is required. However, casting between incompatible types leads to run-time exceptions. PAT supports multidimensional arrays by converting them into one-dimensional arrays. The index range of an array dimension may be specified explicitly by giving the lower bound or the upper bound or both. Examples are as follows:

```
var x = 0; // variable x set to 0
var ba = [1, 2, 3, 4]; // array ba with 4 elements
var leader[3]; // array leader with 3 elements set to 0
var knight : {0..} = 0; // array knight with specified lower index 0
```

Elements of an array may be set using event-prefixing, e.g.,

```
P() = a {m[1][9] = 1} ->  Skip
```

User-defined type may be specified using the declaration **var<***type***>** *var_name***;** e.g.,

```
var<MyType> x; // default constructor MyType is called
var<MyType> x = new MyType(100); // constructor with one parameter
```

Channels and channel arrays are specified using the keyword **channel**, which has two parameters, namely the channel name and the corresponding buffer size. Examples are as follows:

```
channel c 10; // channel c with buffer size 10
channel c[3] 10; // channel array c comprising 3 channels
```

Macros may be defined using the C macro directive **#define**, which has two parameters, namely the macro (instruction) name and its definition. An example is

```
#define condition x==0; // the name is condition, the definition is x==0
```

Model inclusion: a submodel may be included in the current model using the directive **#include**. For example:

```
#include "c:\submodel.csp";
```

5.3.2.2.2 CSP# Process Definitions

CSP# process definitions include:

- Stop
- Skip
- Event prefixing
- Statement block inside events
- Channel input/output
- Sequential composition
- External/internal choice
- Conditional choice
- Case
- Guarded process
- Interleaving
- Parallel composition
- Interrupt
- Hiding
- Atomic sequence
- Recursion
- Assert

Processes may be defined using the equations of the following format:

$$P(x_1, x_2, \ldots, x_n) = exp$$

where P is a process name, x_1, …, x_n is an optional list of formal parameters, and *exp* is a process expression. A process P may be referenced by the expression:

$$P(y_1, y_2, \ldots, y_n)$$

where y_1, …, y_n are the real parameters (or arguments). Self-recursion and mutual-recursion among processes is normally allowed.

Stop is the deadlock process, whereas **Skip** is a process that immediately terminates.

Even prefixing e –> P describes a process which performs an event e first and then behaves as process P. An event may be in a simple form (just the event name) or in a compound form, such as $event\_name.e_1.e_2$ where e_1 and e_2 are expressions composed from variables, including process

parameters, channel input variables, and global variables. Examples are as follows:

```
VM() = coin -> chocolate -> VM(); // chocolate vending machine
Phil(i) = get.i.(i + 1)%N -> Rest(); // i is a process parameter
```

Event name is an arbitrary (user-defined) string. It may also be a channel name. It cannot be the name of a global variable/constant, a process, a process parameter, or a proposition.

Statement blocks inside events (a.k.a. **data operations**): A statement block {*statements*} may be attached to an event simply using the expression *event_name*{*statements*}, where *statements* may include declarations of local variables and arrays, control flow constructs made using the keywords like **if–then–else** and **while**, references to global variables, C# functions, etc., e.g.,

```
P() = incx{x = x + 1;} -> Stop // increment global variable x
```

The attached statement block is executed atomically (i.e., without interleaving with other processes). On the other hand, an event with an attached statement block may be viewed as a labeled piece of code, which is also sometimes used for constructing counterexamples. A reader should note that here are no per process local variables in CSP#, so processes need to use global variables instead.

Invisible events may be specified using the keyword **tau**, e.g., **tau**{$x = x + 1$;} is equivalent with {$x = x + 1$;}.

Channel input/output is written similar to simple event prefixing. Simple examples are as follows (let *c* be the channel name and *P* be a process expression; imagine channel as a FIFO buffer):

```
c!a.b -> P  // output values of expressions a and b
c?x.y -> P // input values of local free variables x and y
c!10 -> P // output constant 10
c?[x > y]x.y -> P // if x > y input values of x and y
```

We may use an arbitrary number of variables/expressions in channel input/output by separating them with dots ('.'), but we cannot use global variables in channel input expressions. Here is an example of two processes involved in an asynchronous communication over the channel *c*:

```
channel c 1;
P(i) = c!i -> P(i)
Q() = c?x -> a.x -> Q()
System() = P(3) ||| Q()
```

In the example above, communication over channel *c* is asynchronous because the size of the channel is nonzero (it is 1). We may turn this communication into synchronous communication by setting the size of the channel

c to zero (**channel** c 0). Furthermore, we may attach statement blocks to asynchronous/synchronous channel input/output, e.g.,

```
channel c 1; // or channel c 0;
var x = 0;
P() = c!x{x = 1} -> P()
Q() = c?y{x = y} -> Q()
System() = P() ||| Q()
```

The execution sequence in the example above is $c!x$, $(x = 1)$, $c?y$, $(x = y)$. Note that the scope of the channel's input variable (such as x and y above) is after the channel input event and within the enclosing process. Such variables may be referenced in the scope, but cannot be updated.

We should also remember that if channel input expressions evaluate to constants, the process can receive only the matching channel outputs. For example, process $P(i) = (c?i.(i + 1) \rightarrow Skip)$ can receive only the sequence of values i, $(i + 1)$ from the channel c. Furthermore, local free variables used in channel input can be reused again in the next channel inputs.

CSP# also supports channel arrays, for example:

```
channel c[2] 1;
S(i) = c[i]!i - S(i);
R() = c[0]?x -> a.x -> R() [] c[1]?x -> a.x -> R();
System() = (|||i:{0..2}@S(i)) ||| R()
```

Channel operations may be used to query the buffer information of an asynchronous channel. A channel operation is invoked by the static method call: **call**(*channel_operation*, *channel_name*). There are five channel operations:

- **cfull** is a Boolean function that tests weather the buffer if full or not.
- **cepmty** is a Boolean function that tests weather the buffer if empty or not.
- **ccount** is an integer function that returns the number of elements in the buffer.
- **csize** is an integer function that returns the buffer size.
- **cpeek** returns the first element (the head) of the buffer.

Sequential composition *P*; *Q* means that *P* and *Q* execute sequentially. There are three kinds of choices in CSP#:

- The **general choice** *P* [] *Q*, which means that either *P* or *Q* may execute. If *P* performs an event first, it takes control, otherwise *Q* takes control.
- The **external choice** *P* [*] *Q* is resolved by the environment through observation of a visible event (not a tau event). If the first event of both *P* and *Q* are visible, *P* [] *Q* and *P* [*] *Q* have the same result.

- The **internal choice** $P <> Q$ means that either P or Q may execute and the choice is made internally and nondeterministically. Although nondeterminism is normally undesirable, it may be useful in the modeling phase for hiding irrelevant (or unknown) details.

The generalized forms of general/external/internal choices are as follows:

- [] $x : \{1..n\} @ P(x)$ is the generalized form for $P(1)$ [] … [] $P(n)$
- [*] $x : \{1..n\} @ P(x)$ is the generalized form for $P(1)$ [*] … [*] $P(n)$
- <> $x : \{1..n\} @ P(x)$ is the generalized form for $P(1)$ <> … <> $P(n)$

Conditional choices are also supported in CSP#. Besides the **traditional conditional choices** used in programming languages which are based on the keywords **if**, **else**, and **else if**, CSP# introduces more specialized conditional choices, such as the **atomic conditional choice** (**ifa**) and the **blocking conditional choice** (**ifb)**. The formats of conditional choices are as follows:

- **if** (*condition1*) *P* **else if** (*condition2*) *Q* **else** *R*. Of course, the shorter **if** and **if–then–else** formats are also allowed.
- **ifa** (*condition*) *P* **else** *Q*. The atomic conditional choice (**ifa**) performs condition checking and the first event of *P* or *Q* atomically.
- **ifb** (*condition*) *P*. The blocking conditional choice (**ifb**) is similar to the guarded process, but unlike the guarded process, in **ifb** condition checking and process execution are separated. There is no **else** in **ifb**, and side effects are not allowed.

The **case** construct in CSP# is somewhat similar to the **switch–case** construct in say, C#:

case {
  condition1: *P*
  condition2: *Q*
  …
  default: *R*
}

The **guarded process** executes when its guarded condition is satisfied (i.e., the *condition* is **true**), otherwise the whole process waits:

[*condition*] *P*

Interleaving $P \;|||\; Q$ means that P and Q execute concurrently without barrier synchronization, except during termination events (termination events must be executed jointly by all the interleaving processes). Of course,

P and *Q* may communicate over shared variables and channels. The generalized format of interleaving is as follows:

$$||| \ x : \{0..n\} @ P(x)$$

We may specify groups of interleaving processes by using looping variables with a finite, or even infinite, range. We may do the same with the parallel composition and the internal/external choices. Examples are as follows:

```
||| {50} @ P(); // interleaving  of 50 P()
||| {..} @ Q(); // interleaving  of infinite number of Q()
||| {} @ P(); // this is equivalent to Skip
[] x : {0..1} @ ( (||| {x} @ P()) ||| (||| {x} @ Q()) )
// <=> (Skip|||Skip) [] (||| {1} @ P()) ||| (||| {1} @ Q())
```

A looping variable *x* may also be used as a process parameter within the process, e.g.:

```
||| x : {0..n} @ (a.x -> Skip)
```

Generally, the symbols used to define a looping variable's range (like *n* in the example above) can be global constants or process parameters, but they cannot be global variables.

Parallel composition *P* || *Q* means that *P* and *Q* execute concurrently with possible **lock-step synchronization**, a.k.a., **barrier synchronization**. Lock-step synchronization means that *P* and *Q* simultaneously perform the same event. In the following example, *P* and *Q* are lock-step synchronized by the event *b*:

```
P() = a -> b -> Stop;
Q() = b -> Stop;
System() = P() || Q()
```

The execution sequence for the example above is *a*, *b*, *Stop*, because *P* performs *a* first, then both *P* and *Q* perform *b*, and, finally, both *P* and *Q* perform *Stop*. Obviously, lock-step synchronization assumes that the alphabets of parallel processes are known. It is well-known that determining the alphabet of a process automatically is generally not trivial (because of process self/mutual referencing and the usage of process parameters), and, in fact, sometimes it is not even possible (for example, in the case of a nonterminating processes). However, PAT provides a best-effort automatic procedure for determining the **default alphabet** of a given process. When the default alphabet is not as expected, we may manually modify it. For example, if we use data operations (statement blocks attached to events), PAT will not cover them when determining the default alphabet, and we would have to manually modify the default alphabet.

Alternatively, we may decide to manually specify the alphabet of a process using the directive **#alphabet** *P* {*events*}, where *P* is the process name, and *events* is a comma-separated list of event expressions. The event expressions may also contain variables, for example, the alphabet of a process *P*(*x*) may be specified as **#alphabet** *P* {*a.x*};.

The important principle of CSP# (and PAT) is that the process signature comprises both the process expression and the process alphabet, i.e., processes with the same process expression, but with different alphabets, which are seen as different processes. To cope with this, we sometimes need to introduce supplementary processes. For example, if the process *P* has different alphabets in two different parts of a model, we may introduce supplementary processes *Q* and *R*:

```
Q() = P();
#alphabet Q = {x};
R() = P();
#alphabet R = {y};
```

Generalized parallel composition has the following format:

|| *x* : {0..*n*} @ *P*(*x*);

We may also use indexed event lists within alphabets. For example:

```
#alphabet P {x:{0..N}; y:{0..N} @ e.x.y};
```

The **interrupt** composition *P* **interrupt** *Q* means that *P* executes until the first visible event of *Q* is engaged, and then control is switched to *Q* (the first visible event may occur at any point of *P*). The corresponding execution trace is a trace of *P*, followed by a trace of *Q*. The main purpose of interrupt abstraction is modeling the interrupt processing behavior.

Hiding may be used to define a process with a reduced alphabet, e.g., *P* \ *A* is a process whose alphabet is **#alphabet**(*P*) \ *A*, where *A* is any subset of **#alphabet**(*P*). We use hiding to hide unimportant events from a process alphabet (for example, to prevent unwanted synchronization in parallel composition) or in order to introduce nondeterminism. For example *Phil* specifies a philosopher who gets forks, eats, and puts the forks away when finished, whereas *dashPhil* hides the events related to forks:

```
Phil() = getfork.1 -> getfork.2 -> eat ->
         putfork.1 -> putfork.2 -> Phil();
dashPhil() = Phil() \ { getfork.1, getfork.2, putfork.1, putfork.2};
```

For our convenience, we may use indexed event lists for defining a set of events with the same prefix, for example:

```
dashPhil() = Phil() \ { x:{1..2}@getfork.x, y:{1..2}@putfork.y};
```

Atomic process *P* is declared using the declaration **atomic**{*P*}, which means that its events should be executed atomically. Also, if a statement

block is prefixed with the keyword atomic, this block should be executed as one superstep without interleaving with other processes. Such a statement block may contain any process statements and may be nondeterministic. Generally, an atomic process has a higher priority than a non-atomic process, i.e., if an atomic process has an enabled event, that event will execute before the events of non-atomic processes. However, if multiple atomic processes are enabled, they interleave each other.

We may use atomic processes and atomic statement blocks to reduce the model state space and thus speedup model checking, especially when the model comprises parallel process compositions. The state space may be sometimes reduced exponentially. Using **atomic** is actually similar to manual partial order reduction. An important rule is that local events that are invisible to the verifying property and independent of other events will get the higher priority.

Recursion is constructed by the self or mutual process referencing. The following example illustrates a system with mutual recursion:

```
P(i) = a.i - Q(i);
Q(i) = b.i - P(i);
System() = P(1) || Q(2);
```

A parameter of the recursive process may be any valid expression, e.g., *P*(*x* + *y*), *P*(**new** *List*()), etc. However, when a parameter is a user-defined type, the user must take special care to pass the correct value type, because CSP# is not a typed language, so PAT does not support compile time type checking. The user should be also very conscious of possible negative side effects, which may, for example, appear within constructs with choices (e.g., *exp1* [] *exp2*—a side effect in *exp1* may remain even when *exp2* is selected).

We may also use recursion to implement common loops. For example, the behavior of the while loop **while** (*condition*) {*P*()} is equivalent to the behavior of the following process:

```
Q() = if(condition) {P(); Q()};
```

Assert is used to add an assertion in the program. PAT simulator and verifiers check the assertion in run-time, and, if the assertion fails, the corresponding run-time exception is thrown and the system evaluation is stopped. For example:

```
var x = 0;
P() = assert(x = 0); a{x = x + 1;} -> P();
```

5.3.2.2.3 CSP# Assertions

Assertions are queries about system behavior. CSP# assertions include:

- Deadlock-freeness
- Divergence-free

- Reachability analysis
- Linear Temporal Logic (LTL)
- Refinement/Equivalence

The deadlock-freeness assertion claims that a process *P* is deadlock-free:

#assert *P*() **deadlockfree**;

PAT uses a depth-first-search or a breadth-first-search algorithm to search the process's state space. A **deadlock state** is a state with no further movement, except a **successfully terminated** state. A process is **deadlock-free** if it does not have any deadlock states.

A **divergence-free** assertion claims that a process *P* is divergence free:

#assert *P*() **divergencefree**;

A process is **divergent** if it performs internal transitions forever without engaging in any useful events, e.g., $P = (a \text{ -> } P) \setminus \{a\}$; a **divergent-free** process is a process that is not divergent.

A **deterministic** assertion claims that a process *P* is deterministic:

#assert P() **deterministic**;

A process is **deterministic** if it does not have a state with more than one outgoing transitions driven with the same event. Otherwise it is **nondeterministic**, e.g., *P* = *a* –> *Skip* [] *a* –> *Stop*.

The **nonterminating** assertion claims that a process *P* is nonterminating:

#assert *P*() **nonterminating**;

A process is **nonterminating** if it does not have a terminating state (either a successfully terminated state or a deadlock state), e.g., such as *P* = *a* –> *P*.

The **reachability** assertion claims that a process *P* may reach a state satisfying a given *condition* (where a condition is a proposition defined as a global definition):

#assert *P*() **reaches** *condition*;

In the following example, the reachability assertion claims that the process *P* reaches a state satisfying the condition ($x < 0$):

```
#define goal x < 0;
var x = 0;
P() = a{x = x + 1;} -> P();
#assert P() reaches goal; // this will not be satisfied
```

The **optimized reachability** allows, for example, minimizing a given *cost* function during the reachability search:

#assert *P*() **reaches** *goal* with min(*cost*);

For example:

```
#define goal x = 14;
var cost = 0;
var x = 0;
P() = if (x <= 14) {{some arithmetic with x} -> P()};
#assert P() reaches goal with min(cost);
```

LTL assertion claims that a process *P* satisfies a given LTL formula *F*:

#assert *P*() |= *F*;

An LTL formula is evaluated on an infinite sequence of truth evaluations over a path traversing the process state space, and a specified position on that path. The syntax of an LTL formula *F* is as follows:

F = *event* | *proposition* | **[]** *F* | **<>** *F* | **X** *F* | *F1* **U** *F2* | *F1* **R** *F2*

where **[]** (or '**G**') reads as **always** , **<>** (or '**F**') reads as **eventually**, **X** reads as **next**, **U** reads as **until**, and **R** (or '**V**') reads as **release** (note that in PAT **[]**, **<>**, **R** may also be written as '**G**', '**F**', and '**V**', respectively).

The semantic of unary modal operators is as follows:

- **X** ϕ, ne**X**t, ϕ holds in the next state: • → • ϕ - - - → • → •
- **G** ϕ, **G**lobally, ϕ holds on the entire subsequent path: • ϕ → • ϕ - - - → • ϕ → • ϕ
- **F** ϕ, **F**inally, ϕ eventually has to hold: • → • - - - → • ϕ → • (holds somewhere on the subsequent path)

The semantic of binary modal operators is as follows:

- **U** ϕ, **U**ntil, ϕ holds at the current or future position, and ψ has to hold until that position; at that position ψ does not have to hold anymore: • ψ → • ψ - - - → • ψ → • ϕ
- **R** ϕ, **R**elease, ϕ is true until the first position in which ψ becomes true, or ϕ is true forever if such position does not exist: • ϕ → • ϕ - - - → • ψ → • ψ, or • ϕ → • ϕ - - - → • ϕ → • ϕ

The LTL assertion is true if and only if the given formula *F* is satisfied for all the possible paths corresponding to all the possible system executions.

Internally, PAT first constructs a Buchi automaton equivalent to the negation of *F* and a Buchi automaton of the process *P*, and then uses these automatons to check the LTL assertion. For example, the following LTL assertion claims that the philosopher can always eventually eat, i.e., the **nonstarvation property**:

```
#assert Phil() |= [] <> eat;
```

Events in LTL formulas may also be component events like *eat*.0, and channel events like "c!3.8" and "c?19" (here we must use "" because '!' and '?' are special characters). In case of synchronous channels, PAT automatically converts channel input/output operators ('!' and '?') to dots in the events, e.g., the channel event "c!3.8" is converted to c.3.8.

Refinement/Equivalence is the FDR (Failures-Divergences Refinement) approach for checking whether an implementation meets its specification. In contrast to an LTL assertion, a **refinement assertion** compares the complete behaviors of two processes, for example, whether one is a subset of another. CSP# supports the following notions of a refinement relationship:

- **#assert** *P*() **refines** *Q*(): *P*() refines *Q*() in the trace semantics
- **#assert** *P*() **refines <F>** *Q*(): *P*() refines *Q*() in the stable failures semantics
- **#assert** *P*() **refines <FD>** *Q*(): *P*() refines *Q*() in the failures divergence semantics

When it comes to verifying CSP# assertions, PAT supports the following options of admissible behavior (the process-level options are enabled only for the systems with interleaving or parallel composition):

- All (or No Special Fairness) is a default option that allows all behaviors to occur. We choose this option to give all the next states (that have the same previous state) the same fairness, i.e., the same possibility to happen, which also means that there is no special fairness for each process.
- Event-level Weak Fair Only means that for every event in the system, if the event is eventually always enabled, then the event always eventually occurs.
- Event-level Strong Fair Only means that for every process in the system, if the process is always eventually enabled, then the event always eventually occurs.
- Process-level Weak Fair Only means that for every process in the system, if the process is eventually always enabled, then the process is always eventually engaged.

- Process-level Strong Fair Only means that for every process in the system, if the process is always eventually enabled, then the process is always eventually engaged.
- Global Fair Only (or Strong Global Fairness) means that for every transition in the system, if the transition can always eventually be taken, then the transition is actually always eventually taken.

A detailed discussion of the above listed options is outside the scope of this book. Also, in order to save the space in the following examples, we sometimes present verification results for only some of the options. Shorter counterexamples are provided without comment so that the reader may analyze and think about them, while longer counterexamples are skipped to save space (of course, an interested reader may repeat the presented experiments using the freely available PAT and reproduce all the counterexamples on their own).

5.3.2.3 Examples of Formal Verification Based on CSP# and PAT

In this section, we study the following examples:

- Alternating bit protocol
- Two-phase commit protocol
- Various leader election protocols in the complete graphs, the rings, and the rooted directed trees
- Telecomm service system

5.3.2.3.1 Alternating Bit Protocol

Alternating Bit Protocol (ABP) is a data link layer protocol that retransmits lost or corrupted messages. Actually, it is a special case of a sliding window protocol where a timer regulates the order of messages to provide reliable message transmission over a data link, using the 1-bit window. Transmitter **A** sends messages to receiver **B** (initially the channel from **A** to **B** is empty). Each message contains data and a 1-bit **sequence number** (SN) whose value is 0 or 1. **B** acknowledges the successfully received messages by sending the appropriate ACK: ACK0 for a message with SN 0 or ACK1 for a message with SN 1.

A resends a message continuously with the same sequence number until it receives an ACK with the same sequence number, then **A** toggles (complements) the sequence number and starts sending the next message. Symmetrically, when **B** receives an uncorrupted message with SN 0, **B** resends ACK0 continuously until it receives an uncorrupted message with SN 1, then it switches to ACK1, etc. Therefore, **A** may still receive ACK0 when

it has already switched to resending a message with SN 1, and vice versa. **A** treats such ACKs as negative-ACKs (NAKs) by simply ignoring them.

The ABP is initialized by sending a bogus message and ACKs with SN 1, so the first real message is a message with SN 0.

We illustrate the ABP in Figure 5.6. **A** starts by sending the information message I1 with the data bit 1, and it keeps resending it until it receives ACK1. Once **B** receives the message, it acknowledges it by the ACK1, and it keeps resending ACK1 until it receives the message I0. In order to keep the figure readable, we show the message I1 and ACK1 as quarter-length arrows. Later on, when **A** receives ACK1 it starts sending I0, when **B** receives I0 it starts sending ACK0, and so on. In order to keep the figure readable, we do not show other messages that were resent.

The parametrized ABP model in CSP# was created by Dr. Sun Jun. The following is ABP model for the parameter *ChannelSize* set to 1 (in the model that has the constant *CHANNELSIZE*):

```
#define CHANNELSIZE 1;
channel c CHANNELSIZE;
channel d CHANNELSIZE;
channel tmr 0;

Sender(alterbit) =
(c!alterbit -> Skip [] lost -> Skip);
tmr!1 -> Wait4Response(alterbit);

Wait4Response(alterbit) =
(d?x -> ifa (x == alterbit) {
          tmr!0 -> Sender(1 - alterbit)
        } else {
          Wait4Response(alterbit)
        })
[] tmr?2 -> Sender(alterbit);

Receiver(alterbit) =
c?x -> ifa (x == alterbit) {
         d!alterbit -> Receiver(1 - alterbit)
       } else {
         Receiver(alterbit)
       };

Timer = tmr?1 -> (tmr?0 -> Timer [] tmr!2 -> Timer);

ABP = Sender(0) ||| Receiver(0) ||| Timer;

#assert ABP deadlockfree;
#assert ABP |= []<> lost;
```

In the model above we model the sender (i.e., transmitter), the receiver, and the sender's timer by the processes *Sender* (and *Wait4Response*), *Receiver*, and *Timer*, respectively. For simplicity, messages only have the sequence number and no data. Messages from *Sender* to *Receiver* are transferred over the channel *c*, whereas messages from *Receiver* to *Sender* are transferred over the channel *d*. Both channels *c* and *d* are ordinary CSP# channels, but in this

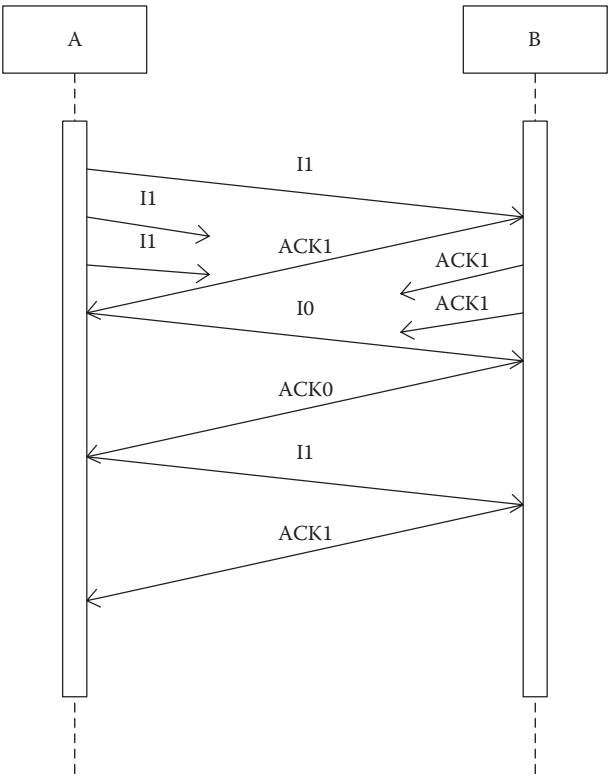


FIGURE 5.6
ABP sequence diagram.

model, we call them the unreliable channel *c* and the perfect (or reliable) channel *d*, because the former models an unreliable channel and the latter models a reliable channel. The third channel in this model is a synchronous channel *tmr* between *Sender* and *Timer*, which is used to model timer-related events, namely start timer (the event *tmr*.1), stop timer (the event *tmr*.0), and timeout (the event *tmr*.2).

As mentioned above, the sender is modeled by the processes *Sender* and *Wait4Response*. Within *Sender*, we use external choice **[]** to model the unreliability of channel *c*. In particular, the construct (*c*!*alterbit* -> *Skip* **[]** *lost* -> *Skip*) means that *Sender* will either successfully send the message *alterbit*, or it will skip sending it, which is equivalent to losing the message on an unreliable channel—in the model, this case corresponds to the event *lost*. Next, *Sender* starts the timer using the channel output event *tmr*!1, and, further on, behaves as *Wait4Response*.

Within *Wait4Response* we use external choice **[]** to model a possible timer expiration event (i.e., the timeout behavior). In particular, *Wait4Response* will

either receive an ACK/NAK from the *Receiver* (the event *d*?*x* when the *alterbit* sent from *Receiver* will be assigned to the local variable *x*) or the timeout will occur (the event *tmr*?2). After receiving an ACK/NAK, *Wait4Response* atomically compares its *alterbit* with *x*, which is the *alterbit* sent by *Receiver*. If these values are equal (it means that *Wait4Response* received an ACK), then *Wait4Response* stops the timer (by using the event *tmr*!1), creates the new message by toggling its *alterbit* (by using simple arithmetic expression: 1 – *alterbit*), and after that behaves as *Sender*.

If the value of *Wait4Response*'s *alterbit* and *x* are not equal (it means that *Wait4Response* received NAK), then *Wait4Response* ignores that NAK (just does noting), and continues waiting for ACK, i.e., continues behaving as *Wait4Response*. In case of timeout (the event *tmr*?2), *Wait4Response* further on behaves as *Sender*.

Receiver operates symmetrically to *Sender* (and *Wait4Response*). When *Receiver* receives an uncorrupted or corrupted message (the event *c*?*x*), it atomically compares its *alterbit* with *x*, which is the *alterbit* sent by *Sender*. If these values are equal (it means that *Receiver* received an uncorrupted message), then *Receiver* sends ACK (i.e., its current *alterbit*), constructs the new ACK for the next uncorrupted message (by using the simple arithmetic expression: 1 – *alterbit*) and behaves as *Receiver*. Alternatively, if *Receiver* received a corrupted message (where the values of its current *alterbit* and *x* where not equal), it continues waiting an uncorrupted message, i.e., it continues behaving as *Receiver*.

The *timer* models a discrete timer. At the beginning *Timer* waits to be started (the event *tmr*?1). Once started, *Timer* may be either ([]) stopped by *Wait4Response* (the event *tmr*?0) or it may expire and generate a timeout signal (the event *tmr*!2) towards *Wait4Response*. In both cases, after engaging in a prefix event (*tmr*?0 or *tmr*!2) it continues behaving as *Timer*.

The complete system is modeled as a parallel composition of *Sender*, *Receiver*, and *Timer*. Initially, both *Sender* and *Receiver* set their local variables *alterbit* to 0.

There are two assertions at the end of the model. The first assertion claims that ABP is deadlock-free. As a result of verifying this assertion, PAT produces the following positive report:

```
The Assertion (ABP() deadlockfree) is VALID.
```

The second assertion claims that always, at some point, a message from *Sender* to *Receiver* will be lost (the event *lost* will happen). The verification result for this assertion depends on the admissible behavior option that we select. As expected, if we select the options "Event-level Strong Fair Only" or "Global Fair Only", the verification result is positive:

```
The Assertion (ABP() |= []<> lost) is VALID.
```

But, if we select the option "All", the result is negative with the following counterexample:

```
The Assertion (ABP() |= []<> lost) is NOT valid.
A counterexample is presented as follows.
<init -> c!0 -> (c?0 -> d!0 -> tmr.1 -> d?0 -> tmr.0 -> c!1 -> c?1
-> d!1 -> tmr.1 -> d?1 -> tmr.0 -> c!0)*>
```

Similarly, if we select the options "Event-level Weak Fair Only," "Process-level Weak Fair Only," or "Process-level Strong Fair Only," the result is also negative, with rather lengthy counterexamples that a reader may reproduce on their own.

5.3.2.3.2 Two-Phase Commit Protocol

Two-phase commit protocol (2PC) is one of the most widely used atomic commitment protocols (ACPs). It coordinates all the processes participating in a distributed atomic transaction on whether they should **commit** or **abort** (or **rollback**) the transaction. In the theory of distributed computing, 2PC is viewed as a specialized **consensus protocol**. The 2PC advantages are simplicity and resilience to many temporary system failures, such as process, network node, or communication failures. However, in some rare cases, system administrators must perform manual failure recovery procedures. To enable failure recovery, which is automatic in most of the cases, participating processes must maintain logs of the protocol's states. Many existing 2PC variants use different logging strategies and recovery procedures.

The protocol relies on the following three assumptions:

- One node is the coordinator, whereas the rest of the nodes are participants (the coordinator may be selected using a leader election protocol).
- Each node has a stable storage for storing a write-ahead log, which is never lost or corrupted in a node crash.
- No node crashes forever.
- Any two nodes can (directly or indirectly) communicate with each other.

During **normal operation** (i.e., when there are no failures) the protocol consists of the following two phases:

- The **commit request phase** (or **voting phase**), in which the process **coordinator** requests from all the processes participating in the transaction (or **participants**, **cohorts**, **workers**, or **pages**) to prepare to commit/abort the transaction by performing all the necessary steps locally, and to vote "yes" (**commit**) if the local preparation was successful or "no" (**abort**) if some problem during local preparation was detected.

- The **commit phase**, in which the coordinator decides whether to commit (if all the participants voted "yes") or abort the transaction (if at least one participant voted "no"), and notifies the decision result to all the participants. The participants, in turn, perform all necessary local actions (effectively realizing commit) on their local resources (or **recoverable resources**) and their portions in the transaction's output (if any).

The commit request phase consists of the following steps:

1. The coordinator sends the message **query to commit** to all the participants and waits until it receives replies from all of them.
2. The participants execute the transaction locally (and update their logs) to the point where they will be asked to commit/abort.
3. Each participant replies to the coordinator with the message **agreement**, which carries its vote—**yes** (commit) if its actions were successful, or **no** (abort) if otherwise.

The completion phase in case of success (commit) consists of the following steps:

1. The coordinator sends the message **commit** to all the participants and waits for their **ACK**s.
2. Each participant completes the transaction locally and releases all locks and resources.
3. Each participant sends the message **ACK** to the coordinator.
4. The coordinator completes the transaction once it receives all the **ACK**s.

The transaction will fail if any of the participants votes **no**, or the coordinator's timer expires (and signals a timeout). The completion phase in case of failure (abort) consists of the following steps:

1. The coordinator sends the message **rollback** to all the participants and waits for their **ACK**s.
2. Each participant undoes the transaction locally, and then releases all locks and resources.
3. Each participant sends the message **ACK** to the coordinator.
4. The coordinator completes the transaction once it receives all the **ACK**s.

We illustrate the 2PC by the sequence diagram in Figure 5.7. As shown, the protocol consists of two phases. In the commit request phase, the coordinator

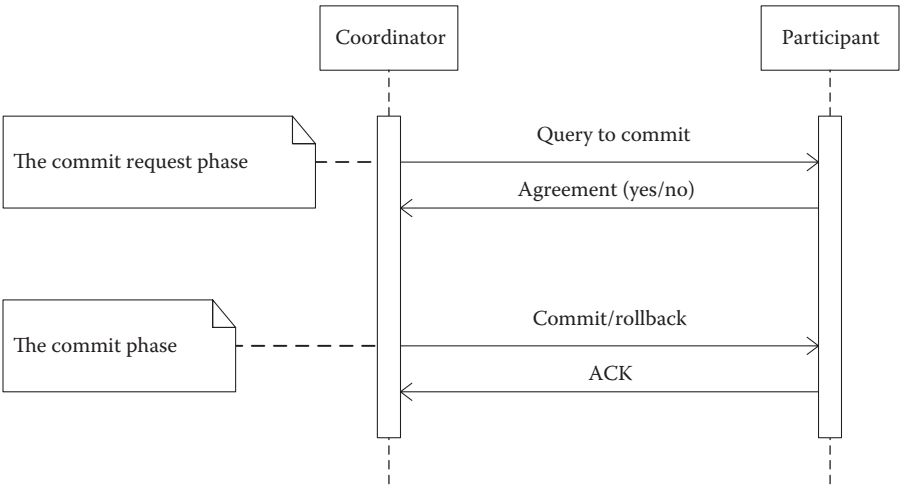


FIGURE 5.7
2PC sequence diagram.

sends the message **query to commit** to each participant, and all the participants vote yes or no by using the corresponding message **agreement**. In the commit phase, depending on the results of the previous phase, the coordinator sends either the message **commit** or the message **rollback** to each participant, and all the participants reply with the message **ACK**.

The parametrized 2PC model in CSP# was created by Dr. Sun Jun slightly different than the one explained above. The following is the simplified 2PC model for the parameter *Page* set to 2 (in the model that is the constant *N*):

```
#define N 2;
enum {Yes, No, Commit, Abort};
channel vote 0;
var hasNo = false;

//The following models the coordinator
Coord(decC) =
(|||{N}@ request -> Skip);

(|||{N}@ vote?vo -> atomic{tau{if (vo == No) {hasNo = true;}} -> Skip});

decide -> (
  ([hasNo == false] (|||{N}@inform.Commit -> Skip);
    CoordPhaseTwo(Commit))
  []
  ([hasNo == true] (|||{N}@inform.Abort -> Skip);
    CoordPhaseTwo(Abort))
);

CoordPhaseTwo(decC) = |||{N}@acknowledge -> Skip;

//The following models a page
Page(decP, stable) =
```

```
request -> execute ->
  (vote!Yes -> PhaseTwo(decP) [] vote!No -> PhaseTwo(decP));

PhaseTwo(decP) =
inform.Commit -> complete -> result.decP -> acknowledge -> Skip
[]
inform.Abort -> undo -> result.decP -> acknowledge -> Skip;

#alphabet Coord {request, inform.Commit, inform.Abort, acknowledge};
#alphabet Page {request, inform.Commit, inform.Abort, acknowledge};

System = Coord(Abort) || (|||{N}@Page(Abort, true));

Implementation =
System \ {request, execute, acknowledge, inform.Abort, inform.Commit,
  decide, result.Abort, result.Commit};

Specification = PC(N);
PC(i) =
[i == 0](|||{N}@complete -> Skip)
[]
[i > 0](vote.Yes -> PC(i-1) [] vote.No -> PU(i-1));

PU(i) =
[i == 0](|||{N}@undo -> Skip)
[]
[i > 0](vote.Yes -> PU(i-1) [] vote.No -> PU(i-1));

#assert System deadlockfree;
#define has hasNo == 1;
#assert System |= [](has -> <> undo);
#assert System |= [](request -> <> undo);

#assert Specification deadlockfree;
#assert Implementation refines Specification;
```

In the model above, we model the coordinator and the participant (page) by the processes *Coord* (and *CoordPhaseTwo*) and *Page* (and *PhaseTwo*), respectively. For simplicity, only messages carrying a *Page*'s vote (*Yes*/*No*) are sent over the channel *vote* to *Cord*. The rest of the communication is modeled as a barrier synchronization, mostly using component events like *inform*. *Commit*, where *inform* corresponds to a channel and *Commit* corresponds to a message. In the two special cases, simple events are used rather than component events, namely the event *request* models the exchange of a **query to commit** message, whereas the event *acknowledge* models the exchange of an **ACK** message. This mapping of message names (given in the informal protocol specification at the beginning of this section) to the corresponding events used in the CSP# model was done with a good choice of event names, so that the reader would not have any difficulties in recognizing the correspondences.

At the beginning of the model, we define the global constants *N*, *Yes*, *No*, *Commit*, and *Abort*; the channel *vote* with the (FIFO buffer) size 0 (which implies synchronous communication); and the Boolean variable *hasNo* with the initial value **false** (assuming final success, i.e., commit).

The process *Coord* initially executes the event *request* once per each *Page* in the system (here twice, because *N*==2): (|||{*N*}@ *request* –> *Skip*); and then *Coord* waits for the votes from all *Page*s: |||{*N*}@ *vote***?***vo*. After receiving a vote from a *Page*, *Coord* atomically (see the keyword **atomic**) checks whether the vote (in the variable *vo*) is *No*, and if it is, *Coord* sets *hasNo* to **true** (effectively changing the final result to failure, i.e., abort); otherwise it ignores the vote (*Skip*).

Next, *Coord* decides the final outcome (success/failure) based on the contents of the variable *hasNo* and informs the pages accordingly. In particular, it first executes the observable event *decide* and then executes either the event *inform.Commit* (if *hasNo* is **true**), or *inform.Abort* (otherwise), once per each *Page* in the system: |||{*N*}@*inform.Commit* or |||{*N*}@*inform.Abort*. Further on, *Cord* behaves as *CoordPhaseTwo* wherein it simply ignores the event *acknowledge* (the reader should note that *CoordPhaseTwo* corresponds only to the points 3 and 4 in the informal specification of the second phase of 2PC, given at the beginning of this section).

Page initially synchronizes with *Coord* using the event *request* and executes the externally observable event *execute* (which models local transaction processing). The local page's actions may be either successful or unsuccessful, and we model this possibility using the external choice operator **[]** (remember, we used **[]** similarly in the model of ABP in the previous section). Next, *Page* sends its vote to *Coord* – *Yes* (if local processing was successful) or *No* (otherwise). Further on, *Page* behaves as *PhaseTwo* (which corresponds to the page's side of the second phase of 2PC). It is important to notice that *Page* passes the value *Commit/Abort* (if its vote was *Yes/No*) to *PhaseTwo* using the process parameter *decP* (decision of a *Page*).

PhaseTwo's actions depend on the notification from *Coord*. In case the notification was *inform.Commit*, *PhaseTwo* sequentially executes the events *complete* (which models successful commit), *result.decP* (which is in this case equal to *result.Commit*), and *acknowledge*. Similarly, in the case that the notification was *inform.Abort*, *PhaseTwo* sequentially executes the events *undo* (which models abort), *result.decP* (which is, in this case, equal to *result.Abort*), and *acknowledge*.

Next, we define alphabets of *Coord* and *Page* (they are equal) by listing the events that are used for barrier synchronization between them, namely, *request*, *inform.Commit*, *inform.Abort*, and *acknowledge*. The complete *System* is defined as a parallel composition of *Coord* and interleaving of *Page*s:

System = *Coord*(*Abort*) || (|||{*N*}@*Page*(*Abort*, *true*));

In this example, we also demonstrate usage of a refinement assertion. Therefore, we firstly define the process *Implementation* as *System* without all the events related to internal operation of *System*. More precisely, *Implementation* inherits only the events *complete*, *undo*, *vote.Yes*, and *vote.No* from *System*.

Second, we define the process *Specification* as the process *PC*(*N*), where the process *PC*(*i*), in turn, is defined as a mutual recursion of itself and the process *PU*(*i*). A reader may easily see that *PC*(*i*) and *PU*(*i*) are essentially countdown processes, where PC counts down *Yes* votes, whereas *PU* counts down both *Yes* and *No* votes starting with the first *No* vote. If the parameter *i* during counting down of votes reaches the value *i*==0 within the process *PC*(*i*), *PC*(*i*) will execute the *N* instances of the event *complete*; otherwise *PU*(*i*) will finally execute the *N* instances of the event *undo*.

At the end of the model, we define five assertions—three of them are related to *System*, one is related to *Specification*, and the fifth is a refinement assertion. The *System*-related assertions are the following:

- *System* is deadlock free.
- *System* satisfies that always after the point when the condition *has* holds (i.e., *has* is a macro which is defined as *hasNo* == 1, i.e. **true**), the event *undo* will be eventually executed.
- *System* satisfies that always after the point when the event *request* was executed, the event *undo* will be eventually executed.

When these assertions are verified by PAT, as expected, PAT reports that the first two are valid, whereas the third is invalid. The third assertion is invalid because after the initial execution of the event *request*, the resulting event may be either *undo* or *complete*, and not always *undo* as claimed. Here is the counterexample produced by PAT:

```
The Assertion (System() |= [](request-><> undo)) is NOT valid.
A counterexample is presented as follows.
<init -> request -> request -> execute -> vote.Yes -> τ ->
execute -> vote.Yes -> τ -> decide -> inform.2 -> inform.2 ->
complete -> result.Abort -> acknowledge -> complete ->
result.Abort -> acknowledge -> terminate>
```

The *Implementation*-related assertion claims that it is deadlock free. The fifth, and the last assertion in this example claims that *Implementation* **refines** *Specification*. When these two assertions are verified by PAT, as expected, PAT reports that both are valid.

5.3.2.3.3 Leader Election in Complete Graphs

Generally, **leader election** is a fundamental problem in distributed systems, because many hard-distributed problems are easy to solve once a central coordinator is available. An attractive approach to solve the leader election is by using **self-stabilizing algorithms**, which do not require initialization in order to operate correctly, and which can recover from transient faults that may destroy the system state information. Also, among many models, a **network of finite-state anonymous agents** is a rather interesting one, because it models many distributed systems of identical, simple computational nodes,

such as wireless sensor networks, etc. It is well-known that the self-stabilizing leader election is impossible without a **failure detector**, which is a kind of oracle that provides some information to the system that it is unable to compute on its own.

Therefore, Fischer and Jiang (2006) introduced the **eventual leader detector** Ω?. We may imagine Ω? as a black box that provides global status information about the protocol, in particular, whether or not there is a leader in the system. This detector is **weak** in the sense that it does not respond to status changes immediately, but with some indeterminate delay, and it does not report its findings to all the processes (agents) simultaneously. (So some agents may discover status changes sooner than the other processes.) Formally, Ω? provides a Boolean input to each process at each step, such that the following conditions are satisfied by every execution *E*:

- If all, except finitely many, configurations of *E* lack a leader, then all processes receive **false** in all, except finitely many, steps.
- If all, except finitely many, configurations of *E* have one or more leaders, then all processes receive **true** in all, except finitely many, steps.

Thanks to its weakness, Ω? may be simply implemented using timeouts. Each leader periodically sends a keep-alive message, whereas each agent restarts its timer after receiving such a message, and sets the leader detector flag to **true** (indicating that leader is present). On timeout, the process sets the leader detector flag to **false** (indicating that leader is absent). Of course, in an adverse environment, Ω? may temporarily produce incorrect information. However, eventually after the environment stabilizes, Ω? will produce correct information.

Further on in this section, we introduce, model, and analyze the self-stabilizing leader election algorithm for complete graphs (see the example of the complete graph with five nodes in the Figure 5.8) using Ω? (Fischer, 2006), which works under either local or global fairness condition. According to this algorithm, each node has a memory slot that can hold either a leader mark "x" or nothing "-" for a total of two states. Each node receives its current input **true** (T) or **false** (F) from Ω?. A nonleader becomes a leader, when Ω? signals the absence of a leader, and the responder is not a leader. When two leaders interact, the responder becomes a nonleader. Otherwise, no state change occurs.

The algorithm can be formally described by the three pattern rules, which are matched against the state and the input of the initiator and the responder, respectively. If the match succeeds, the states of the two interacting nodes are replaced by the respective states on the right-hand side of the rule. According to the **star convention**, "*" is a symbol that always matches the slot or the input. On the rule's right-hand side, "*" specifies that the contents of the corresponding slot do not change. If no explicit rules match, neither node changes state (i.e., a null transition takes place).

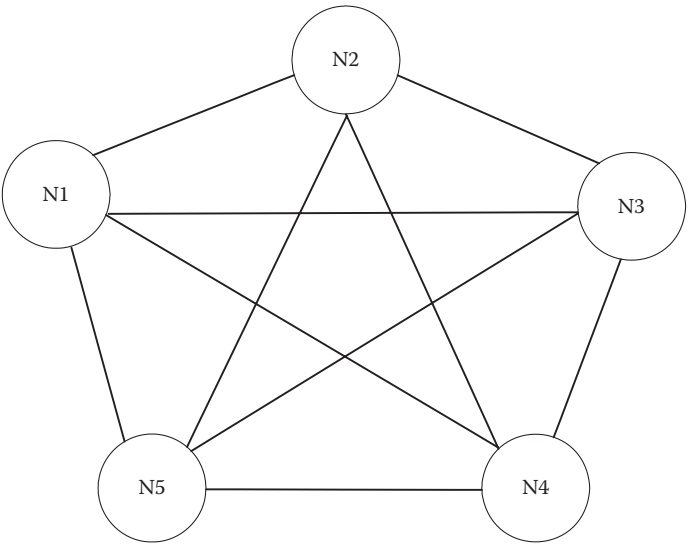


FIGURE 5.8
Example of the complete graph with five nodes.

The three pattern rules are as follows:

Rule 1: ((x, *), (x, *)) –> ((x), (-))
Rule 2: ((-, F), (-, *)) –> ((x), (-))
Rule 3: ((-, T), (-, *)) –> ((-), (-))

The parametrized model of a leader election algorithm for complete graphs in CSP# was created by the PAT Team. The following is the particular model for the parameter *Number of Processes* set to 3 (in the model that is the constant *N*):

```
#define N 3;
var dok = 0;  // detector correct (ok)
var detector = false;
var leader[N];

/*Rule 1*/
Rule1(i, r) =
[leader[i] == 1 && leader[r] == 1]
  (rule1.i.r{leader[r] = 0;} -> Rule1(i, r));

/*Rule 2*/
Rule2(i, r) =
[leader[i] == 0 && leader[r] == 0 && !(( dok == 0 && detector) ||
(dok != 0 && ( leader[0] + leader[1] + leader[2] > 0)))]
  (rule2.i.r{leader[i] = 1;} -> Rule2(i, r));
```

```
/*Rule 3*/
Rule3(i, r) =
[leader[i] == 0 && leader[r] == 0 && ((dok == 0 && detector) ||
(dok != 0 && (leader[0] + leader[1] + leader[2] > 0)))]
  (rule3.i.r -> Rule3(i, r));

// eventual leader detector
DetectorCorrect() =
[dok == 0]
  (progress{dok = 1;} ->  DetectorCorrect());
// detector
RandomDetector() =
[dok == 0]
  ((random1{detector = false;} -> RandomDetector()) []
   (random2{detector = true;} -> RandomDetector()));

Initialization() =
((tau{leader[0] = 0;} -> Skip) [] (tau{leader[0] = 1;} -> Skip));
((tau{leader[1] = 0;} -> Skip) [] (tau{leader[1] = 1;} -> Skip));
((tau{leader[2] = 0;} -> Skip) [] (tau{leader[2] = 1;} -> Skip));

LeaderElection() =
Initialization();
(DetectorCorrect() ||| RandomDetector() |||

Rule1(0,1)|||Rule1(1,0)|||Rule1(0,2)|||Rule1(2,0)|||
Rule1(1,2)|||Rule1(2,1)|||

Rule2(0,1)|||Rule2(1,0)|||Rule2(0,2)|||Rule2(2,0)|||
Rule2(1,2)|||Rule2(2,1)|||

Rule3(0,1)|||Rule3(1,0)|||Rule3(0,2)|||Rule3(2,0)|||
Rule3(1,2)|||Rule3(2,1));

// The Property
#define oneLeader (leader[0] + leader[1] + leader[2] == 1);
#assert LeaderElection() |= <>[]oneLeader;
```

In the model above, we modeled the three rules: Rule 1, Rule 2, and Rule 3, by the processes *Rule1*, *Rule2*, and *Rule3*, respectively. Each of these processes has two parameters, namely *i* and *r*, where *i* is the index of the initiator node and *r* is the index of the responder node. Next, we model the eventual leader detector by the processes *DetectorCorrect* and *RandomDetector*, the random setup of the node's initial states by the process *Initialization*, and the complete system by the process *LeaderElection*.

At the beginning of the model, we define global constants and variables. The global constant *N* is equal to the number of processes (i.e., 3). The value of the global integer variable *dok* determines whether the information provided by the leader detector is correct (value 1) or not (value 0). Initially, *dok* is set to 0, indicating the presence of transient errors in the environment, and later on it is set to 1, indicating that the environment becomes stable and well-behaved. The value of the Boolean variable *detector* represents the output of the leader detector (**true** if there is a leader in the system), which may be erroneous. The integer array *leader* (of size *N*) corresponds to memory slots at each node, which hold the current state of the node (the value 0

means that the node is not a leader, whereas the value 1 means that the node is a leader).

According to Rule 1, the process *Rule1* checks if both *leader*[*i*] and *leader*[*r*] are set to 1 (i.e., if both initiator and responder are leaders), and if they are, it then sets *leader*[*r*] to 0 (i.e., the responder becomes a nonleader). Further on, it behaves again as *Rule1*. Obviously, the process *Rule1* is a straightforward encoding of Rule 1.

Similarly, *Rule2* and *Rule3* are rather straightforward encodings of Rule 2 and Rule 3, respectively. However, there is one important difference; unlike the process *Rule1*, the processes *Rule2* and *Rule3* behave differently in the first phase of the system evolution, when the environment is unstable (i.e., when the value of the variable *dok* is 0), and in the second phase when the environment becomes stable (i.e., when *dok* is set to 1). In the first phase, the processes *Rule2* and *Rule3* behave as specified by the rules (i.e., Rule 2 and Rule 3, respectively), whereas in the second phase (when *dok* is not equal to 0), *Rule2* and *Rule3* execute if there is at least one leader in the system (when the sum of the elements of the array *leader* is greater than 0).

The process *DetectorCorrect* models the system transition from an unstable to a stable state. Initially, the variable *dok* is set to 0 (indicating an unstable environment). Once the process *DetectorCorrect* sees that the variable *dok* is set to 0, it simply sets it to 1 (indicating a stable environment). Similarly, the process *RandomDetector* models possibly erroneous readings from the leader detector. As long as the variable *dok* is set to 0 (indicating an unstable environment), the process *RandomDetector* randomly sets the variable *detector* to either **true** or **false**, but once *dok* is set to 1, it stops writing to the variable *detector* (so its value stabilizes).

The process *Initialization* is a sequence of three sub processes, where each of the sub processes randomly sets the initial state of the corresponding node (i.e., the corresponding element of the array *leader*) to either 0 (nonleader) or 1 (leader), and then terminates (*Skip*). So, the system may start from any possible combination of node states.

The process *LeaderElection* is the sequential composition of the process *Initialization* and the process that is the interleaving of the processes *DetectorCorrect*, *RandomDetector*, and the process instances of the processes *Rule1*, *Rule2*, and *Rule3*, for all possible combinations of values of their parameters *i* and *r*, i.e., (0, 1), (1, 0), (0, 2), (2, 0), (1, 2), and (2, 1).

At the end of the model, we define the macro *oneLeader* and the one *LeaderElection*-related assertion. The macro *oneLeader* is defined as the equation of the sum of the elements of the array leader (which corresponds to the number of leaders currently present in the system) and the constant 1. Of course, the goal of any leader election protocol is that this number of leaders is finally equal to 1, which means that there is exactly one leader in the system. Therefore, the assertion at the end of the model claims that the process *LeaderElection* eventually always satisfies the goal (i.e., the equation) *oneLeader*.

PAT verification reports are as expected. If we select the admissible behavior to be Global Fair Only or Event-level Strong/Weak Fair Only, the assertion is found to be valid. Alternatively, if we select the admissible behavior option All, the assertion is found to be invalid. Here is the counterexample produced by PAT:

```
The Assertion (LeaderElection() |= <>[] oneLeader) is NOT valid.
A counterexample is presented as follows.
<init -> τ -> τ -> τ -> rule1.2.1 -> rule1.2.0 -> (rule2.1.0 ->
rule1.2.1 -> rule2.1.0)*>
```

5.3.2.3.4 Leader Election in Rings

In this section we introduce, model, and analyze the uniform, self-stabilizing leader election algorithm for rings using Ω? (Fischer, 2006), which requires global fairness (and under local fairness is not feasible). See the example of the ring graph with five nodes in the Figure 5.9.

The algorithm is based on the following assumptions: The ring is directed such that each node has a sense of forward (clockwise) and backward (counterclockwise) directions and every interaction takes place between the initiator and its forward neighbor. Each node can store zero or one of each of three kinds of tokens: a bullet "o", a leader mark "x", and a shield "|", for a total of eight possible states. Corresponding to each kind of token is a slot which is empty if the corresponding token is not present, and full if it is present. An empty slot is denoted by "-" whereas a full slot is denoted by the token. The slots in each node are ordered with the bullet first, leader mark second, and shield third. Extending this to a clockwise ordering of all slots in the ring,

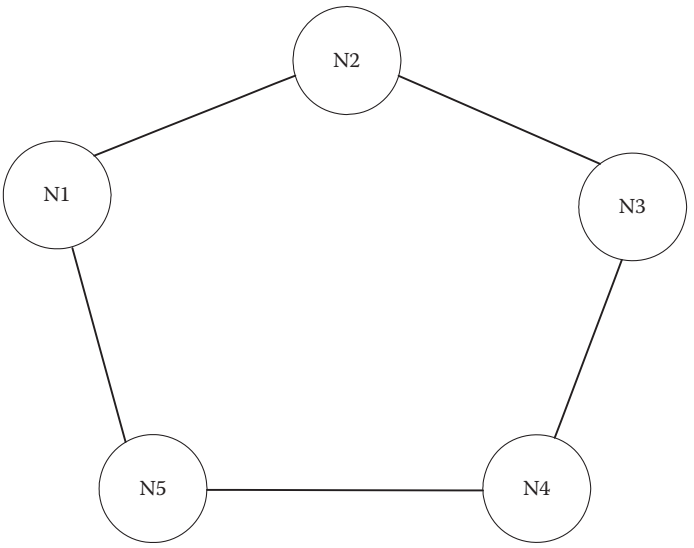


FIGURE 5.9
Example of the ring graph with five nodes.

the shield slot of one node is followed by the bullet slot of the next node in the clockwise order.

The algorithm can be formally described by the five pattern rules:

Rule 1. ((* * *, F), (* * *, *)) –> ((o x |), (* * *))
Rule 2. ((* - |, T), (* * *, *)) –> ((* - -), (- * |))
Rule 3. ((* x |, T), (* * *, *)) –> ((o x -), (- * |))
Rule 4. ((* x -, T), (- * *, *)) –> ((o x -), (- * *))
Rule 5. ((* * -, T), (o * *, *)) –> ((o - -), (- * *))

When two nodes interact and the initiator's input is false (F), a leader and shield are created, and at the same time, a bullet is fired (Rule 1). This is the only way for leaders and shields to be created. When the initiator's input is true (T), the following rules apply: Shields move forward around the ring (Rules 2 and 3), and bullets move backward (Rule 5). Bullets are absorbed by any shield they encounter (Rules 2 and 3) but kill any leaders along the way (Rule 5). If a bullet moves into a node already containing a bullet, the two bullets merge into one. Similarly, when two shields meet, they merge into one. A leader fires a bullet whenever it is the initiator of an interaction (Rules 3 and 4).

In a configuration in which the node *i* has a leader mark, the node *j* has a shield, and all of the slots between *i*'s leader mark and *j*'s shield in clockwise order are empty, the node *i* is called the **protected leader**, and the node *j* is called its **protecting shield**. A node can be both a protected leader and its own protecting shield. The algorithm solves the leader election such that eventually there is exactly one protected leader, one protecting shield, and no unprotected leader.

The parametrized model of the leader election algorithm for rings in CSP# was created by PAT Team. The following is the particular model for the parameter *Number of Processes* set to 3 (in the model that is the constant *N*):

```
#define N 3;
var dok = 0;
var detector = false;
var leader[N];
var bullet[N];
var shield[N];

// Processes
Process(i) =
[!((dok==0 && detector) ||
(dok!=0 && leader[0]+leader[1]+leader[2] > 0))]
  rule1.i.(i+1)%N{bullet[i]=1; leader[i]=1; shield[i]=1;} ->
  Process(i)
[]
[leader[i] == 0 && shield[i] == 1 && ((dok==0 && detector) ||
(dok!=0 && leader[0]+leader[1]+leader[2] > 0))]
  rule2.i.(i+1)%N{leader[i]=0; shield[i]=0; bullet[(i+1)%N] = 0;
  shield[(i+1)%N] = 1;} -> Process(i)
```

```
[]
[leader[i] == 1 && shield[i] == 1 && ((dok ==0 && detector) ||
(dok!=0 && leader[0]+leader[1]+leader[2] > 0))]
  rule3.i.(i+1)%N{ bullet[i] = 1; leader[i] = 1; shield[i] = 0;
  bullet[(i+1)%N] = 0; shield[(i+1)%N] = 1;} -> Process(i)
[]
[leader[i] == 1 && shield[i] == 0 && bullet[(i + 1) % N] == 0
&& ((dok==0 && detector) ||
(dok!=0 && leader[0]+leader[1]+leader[2] > 0))]
  rule4.i.(i+1)%N{ bullet[i] = 1; leader[i] = 1; shield[i]=0;
  bullet[(i+1) % N] = 0;} -> Process(i)
[]
[shield[i] == 0 && bullet[(i+1)% N] == 1 && ((dok==0 && detector) ||
(dok!=0 && leader[0]+leader[1]+leader[2] > 0))]
  rule5.i.(i+1)%N{bullet[i] = 1; leader[i] = 0; shield[i] = 0;
  bullet[(i+1)%N] = 0;} -> Process(i);

// eventual leader detector
DetectorCorrect() =
[dok == 0](progress{ dok = 1;} ->  DetectorCorrect());
//detector
RandomDetector() =
[dok == 0]
  ((guess1{detector = false;} -> RandomDetector())
   []
   (guess2{detector = true;} -> RandomDetector()));

Initialization() =
((tau{leader[0] = 0;} -> Skip) [] (tau{leader[0] = 1;} -> Skip));
((tau{leader[1] = 0;} -> Skip) [] (tau{leader[1] = 1;} -> Skip));
((tau{leader[2] = 0;} -> Skip) [] (tau{leader[2] = 1;} -> Skip));
((tau{bullet[0] = 0;} -> Skip) [] (tau{bullet[0] = 1;} -> Skip));
((tau{bullet[1] = 0;} -> Skip) [] (tau{bullet[1] = 1;} -> Skip));
((tau{bullet[2] = 0;} -> Skip) [] (tau{bullet[2] = 1;} -> Skip));
((tau{shield[0] = 0;} -> Skip) [] (tau{shield[0] = 1;} -> Skip));
((tau{shield[1] = 0;} -> Skip) [] (tau{shield[1] = 1;} -> Skip));
((tau{shield[2] = 0;} -> Skip) [] (tau{shield[2] = 1;} -> Skip));

LeaderElection() =
Initialization();
(DetectorCorrect() ||| RandomDetector() |||
  Process(0)|||Process(1)|||Process(2));

// The Property
#define oneLeader (leader[0] + leader[1] + leader[2] == 1);
#assert LeaderElection() |= <>[]oneLeader;
```

This model is rather similar to the model in the previous section. Actually, some processes are identical or almost identical. The main differences are as follows: In this model, the integer arrays *leader*, *bullet*, and *shield* (each of size *N*), correspond to memory slots at each node, which hold the current state of the node (the element value 1 means that the node holds the corresponding token, whereas the element value 0 means that the node does not hold the corresponding token). All the five pattern rules are encoded within a single process, namely *Process*, rather than being defined as separate processes (as was done in the model in Section 3.2.3.2). *Process* corresponds to a single node within a ring, and its parameter *i* is simply the index of the node in the ring. This index is used to access the corresponding elements of arrays *leader*, *bullet*, and *shield*. Obviously, in order to define the process *LeaderElection* in

this model, we need to make the three *Process*'s instances, namely *Process*(0), *Process*(1), and *Process*(2). Apart from these differences, the models are analogous, and thus the reader should have no difficulties in analyzing the model above, and so we leave it as an individual reader's exercise.

The PAT verification reports are as expected. If we select the admissible behavior option All, the assertion is found to be valid. Alternatively, if we select the admissible behavior to be Global Fair Only or Event-level Strong/Weak Fair Only, the assertion is found to be invalid. Here is the counterexample produced by the PAT for the option Global Fair Only (the other two counterexamples are longer, so we skip them, and leave the reader to reproduce them as an exercise):

```
The Assertion (LeaderElection() |= <>[] oneLeader) is NOT valid.
A counterexample is presented as follows.
<init -> τ -> τ -> τ -> τ -> τ -> τ -> τ -> τ -> τ -> τ -> guess2 -> rule3.0.1 ->
rule3.2.0 -> guess1 -> rule1.1.2 -> rule1.0.1 -> guess2 -> rule5.2.0 ->
rule3.1.2 -> rule3.0.1 -> rule2.2.0 -> guess1 -> rule1.1.2 -> progress ->
rule3.0.1 -> (rule5.2.0 -> rule4.0.1 -> rule5.2.0)*>
```

5.3.2.3.5 Leader Election in Trees

In this section, we introduce, model, and analyze the deterministic, uniform, and self-stabilizing leader election algorithm for rooted directed trees using Ω? (Canepa, 2008), which requires global fairness (like the algorithm in the previous example). See the example of the tree graph with five nodes in Figure 5.10. This algorithm is space optimal because it requires only one memory bit per agent.

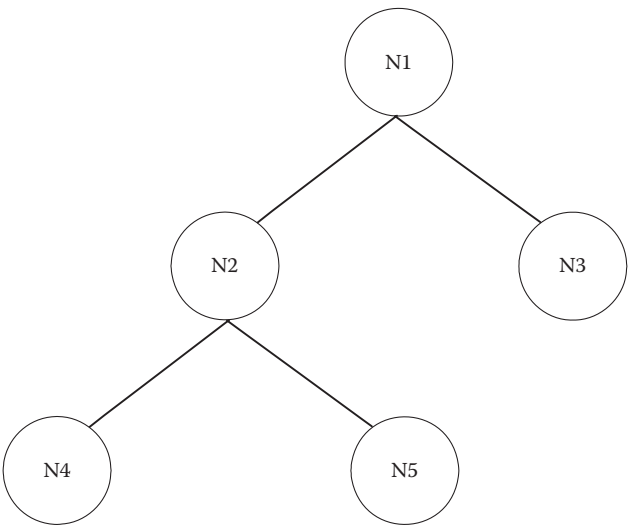


FIGURE 5.10
Example of the tree graph with five nodes.

The algorithm is based on the following assumptions: The root of a rooted directed tree is the only node of in-degree 0, and for each node in the tree, there is a directed path from the root to that node. Each node has a memory slot that can hold either a leader mark "x" or nothing "-", for a total of two states per node, and each node receives its current input true (T) or false (F) from Ω?.

The algorithm can be formally described by the three pattern rules:

Rule 1. ((x, *), (x, *)) –> ((x), (-))

Rule 2. ((-, F), (-, *)) –> ((x), (-))

Rule 3. ((-, *), (x, *)) –> ((x), (-))

Intuitively the algorithm works as follows: A clean agent (i.e., an agent without a leader mark) becomes leader mark holder, when Ω? signals the absence of leader marks, and the responder does not hold a leader mark (Rule 2). When two agents holding a leader mark each interact, the responder becomes clean (Rule 1). If the responder has a leader mark and the initiator is a clean agent, the latter becomes a leader mark holder and the former becomes clean (Rule 3). Otherwise, no state change occurs.

The parametrized model of the leader election algorithm for rings in CSP# was created by the PAT Team. The following is the particular model for the parameter *Number of Processes* set to 3 (in the model that is the constant *N*):

```
#define N 3;
var dok = 0;
var detector = false;
var leader[N];

/*Rule 1*/
Rule1(i, r) =
[leader[i] == 1 && leader[r] == 1]
  (rule1.i.r{leader[r] = 0;} -> Rule1(i, r));

/*Rule 2*/
Rule2(i, r) =
[leader[i] == 0 && leader[r] == 0 && !(( dok == 0 && detector) ||
(dok != 0 && (  leader[0] + leader[1] + leader[2] > 0)))]
  (rule2.i.r{leader[i] = 1;} -> Rule2(i, r));

/*Rule 3*/
Rule3(i, r) =
[leader[i] == 0 && leader[r] == 1]
  (rule3.i.r{leader[i] = 1; leader[r] = 0;} -> Rule3(i, r));

// eventual leader detector
DetectorCorrect() =
[dok == 0]
  (progress{dok = 1;} ->  DetectorCorrect());
// detector
RandomDetector() =
[detectorcorrect == 0]
  ((random1{detector = false;} -> RandomDetector()) []
   (random2{detector = true;} -> RandomDetector()));
```

```
Initialization() =
((tau{leader[0] = 0;} -> Skip) [] (tau{leader[0] = 1;} -> Skip));
((tau{leader[1] = 0;} -> Skip) [] (tau{leader[1] = 1;} -> Skip));
((tau{leader[2] = 0;} -> Skip) [] (tau{leader[2] = 1;} -> Skip));

// The topology is a rooted tree
LeaderElection() =
Initialization();
(DetectorCorrect() ||| RandomDetector() |||

Rule1(0,1)|||Rule1(0,2)|||
Rule2(0,1)|||Rule2(0,2)|||
Rule3(0,1)|||Rule3(0,2));

// The Property
#define oneLeader (leader[0] + leader[1] + leader[2] == 1);
#assert LeaderElection() |= <>[]oneLeader;
```

This CSP# code is completely analogous to the CSP# code given in Section 5.3.2.3.3 (Leader Election in Complete Graphs), so the reader should have no difficulties understanding it. Actually, it uses the same variables and conventions for encoding the pattern rules. Moreover, this example and the example in Section 5.3.2.3.3 use the processes *DetectorCorrect*, *RandomDetector*, and *Initialization*.

The main difference between these two examples is the way how their respective *LeaderElection* processes instantiate the processes *Rule1*, *Rule2*, and *Rule3*. In this section, we introduce the convention that the node with index 0 is the root of the tree, whereas nodes with indexes 1 and 2 are the leaves of the tree. We also assume that the first parameter (*i*) of the processes *Rule1*, *Rule2*, and *Rule3* is the index of the root, whereas the second parameter (*r*) is the index of a leaf. Thus, we create two process instances of each of the processes *Rule1*, *Rule2*, and *Rule3*, for two possible combinations of values for their parameters *i* and *r*, i.e., (0, 1), and (0, 2).

The PAT verification reports are as expected. If we select the admissible behavior to be Global Fair Only or Event-level Strong/Weak Fair Only, the assertion is found to be valid. Alternatively, if we select the admissible behavior option All, the assertion is found to be invalid. Here is the counterexample produced by PAT:

```
The Assertion (LeaderElection() |= <>[] oneLeader) is NOT valid.
A counterexample is presented as follows.
<init -> τ -> τ -> τ -> rule1.0.2 -> random2 -> (random2 -> random2)*>
```

5.3.2.3.6 Telecomm Service System

The Telecomm Service System (TSS) presented in this section is a simplified model of a telephone exchange (a.k.a., Private Branch eXchange, PBX) that supports local calls only. The main PBX call-processing functions are as follows:

- Establishing connections (circuits) between the telephone sets of the two users

- Maintaining such connections as long as the users require them
- Disconnecting those connections as per the user's request
- Providing information for accounting purposes, i.e., metering calls (not modeled in TSS)

Besides these basic functions, PBXs offer many other calling features and capabilities, also known as supplementary services or add-ons. Modeling all of them would result in a rather complex model, thus TSS supports only some of the most frequently used supplementary services, namely the following:

- Call Forward Unconditional (CFU)
- Call Forward when Busy (CFB)
- Originating Call Screening (OCS)
- Originating Dial Screening (ODS)
- Terminating Call Screening (TCS)
- Ring Back When Free (RBWF)

Normally, systems like TSS are designed (and implemented) incrementally. We start with the basic functions, and then add individual, supplementary services incrementally. However, the complex interaction between users and incremental system extensions may lead to unpredictable and undesirable results. Some new features may conflict with each other, or hinder the basic services. Thus, the model of TSS must reflect the high-level design of the system, both the interaction between the users and the compatibility of new services. We also need a comprehensive set of properties covering basic call-processing and the new services in order to verify if all the system requirements are satisfied.

Since the model of TSS is rather complex and there are many properties to verify, we first analyze only the model, and then we introduce and discuss the system properties separately. The parametrized model of TSS in CSP# was created by the PAT Team. The following is the particular model for the parameter *Number of Users*, set to 2 (in the model that is the constant *NoOfUsers*):

```
#define NoOfUsers         2;
#define NoOfChannels      4; // = NoOfUsers+2
#define NIL               3; // = NoOfUsers+1
#define INVALID_USER      2; // = NoOfUsers

// Model variables
var partner=[NIL(NoOfChannels)];
var chan=[NIL(NoOfChannels)];
var connect=[0(NoOfChannels)];
var dev=[1(NoOfChannels)];
var CFU:{0..NIL}=[NIL(NoOfChannels)];
var CFB:{0..NIL}=[NIL(NoOfChannels)];
var RBWF:{0..NIL}=[0(NoOfChannels)];
var lastCall:{0..NIL}=[NIL(NoOfChannels)];
var OCS:{0..NIL}=[NIL(NoOfChannels)];
```

```
var ODS:{0..NIL}=[NIL(NoOfChannels)];
var TCS:{0..NIL}=[NIL(NoOfChannels)];

// Verification variables
var dialNum=[NIL(NoOfUsers)];
var justDial=[0(NoOfUsers)];

System() = (User());
// All users start from Idle
User() = ||id:{0..NoOfUsers-1}@(Idle(id));
#alphabet Idle
  {keepTalking.0.1,stopTalking.0.1,keepTalking.1.0,stopTalking.1.0};

// Processes corresponding to individual call-processing states
Idle(id) =
  idle.id{chan[id]=NIL; partner[id]=NIL; dev[id]=1; connect[id]=0;
  dialNum[id]=NIL; } ->
  ([chan[id]== NIL] dialing.id{dev[id]=0;} -> Dialing(id)
   []
   [chan[id]>= 0 && chan[id] != NIL]
     answerCall.id{partner[id]=chan[id]; dev[id]=0;} -> Answer(id)
   []
   [RBWF[id]==1 && lastCall[id]!=NIL]
   // RBWF feature, reply last call
     if (lastCall[id]!=ODS[id]) {
       ringback.id.lastCall[id]{partner[id]=lastCall[id];
       lastCall[id]=NIL;} -> Calling(id)
     } else {forbidCall.id -> Idle(id)}
  );

Answer(id) =
case{
  partner[id]==TCS[id]:
    ignoreCall.id{if (chan[partner[id]]==id) chan[partner[id]]=NIL;}
    -> Idle(id)

  chan[partner[id]]==id:
    //partner[id] is waiting
    t_alert.id.partner[id] -> T_Alert(id)

  default:
    //chan[partner[id]]!= id -> partner[id] has changed dial number
    partnerChanged.id.partner[id] -> Idle(id)
};

Dialing(id) =
noCall.id -> Idle(id)
[]
([] callId:{0..NoOfUsers}@(
  if (callId!=ODS[id]) {
    dial.id.callId{justDial[id]=1;} ->
    makeCall.id{partner[id]=callId; dialNum[id]=callId;} ->
    Calling(id)
  } else {
    forbidCall.id -> Idle(id)
  }
));

Calling(id) =
if (partner[id]!=OCS[id]) {
  doCall.id.partner[id]{justDial[id]=0;} ->
  case {
    partner[id]==id:
    //id call to itself
```

```
    selfCall.id -> CallBusy(id)

    partner[id]==INVALID_USER:
    //id call to unobtainable user
    invalidCall.id -> CallUnobtainable(id)

CFU[partner[id]] != NIL:
    //partner[id] has CFU add-on
    forwardCall.partner[id].CFU[partner[id]]{
      partner[id]=CFU[partner[id]];} -> Calling(id)

chan[partner[id]]==NIL:
    //partner channel is free
    ringPartner.id{
      chan[partner[id]]=id; chan[id]=partner[id];} -> O_Alert(id)

    chan[partner[id]]!=NIL && CFB[partner[id]]==NIL:
    //partner channel is busy, no CFB add-on
    if (RBWF[partner[id]]==0) {
      //partner does not have RBWF add-on
      busyPartner.id -> CallBusy(id)
    } else { //partner has RBWF add-on
      busyPartner.id{lastCall[partner[id]]=id;} -> CallBusy(id)
                                  }

    chan[partner[id]]!=NIL && CFB[partner[id]]!=NIL:
    //partner channel is busy, has CFB add-on
      forwardCall.partner[id].CFB[partner[id]]{
        partner[id]=CFB[partner[id]];} -> Calling(id)

    default:
    errorCall.id -> ErrorState(id)
  }
} else {
  forbidCall.id{justDial[id]=0;} -> Idle(id)
};

CallBusy(id) =
//id receives busy signal
soundBusy.id -> Idle(id);

CallUnobtainable(id) =
//id calls to unobtainable user
soundInvalid.id -> Idle(id);

O_Alert(id) =
[chan[id]==partner[id] && connect[id]==1]
  callConnect.id.partner[id] -> O_Connected(id)
[]
[chan[id]==partner[id] && connect[id]==0]
  ringOut.id{if (chan[partner[id]]==id) chan[partner[id]]=NIL;
  chan[id]=NIL;} -> Idle(id)
[]
[chan[id]!= partner[id]]
  callStopped.id.partner[id] -> Idle(id)
[]
[chan[id]==partner[id] && connect[id]!= 1 && connect[id] != 0]
  alertError.id -> ErrorState(id);

O_Connected(id) =
[connect[id]==1 && connect[partner[id]]==1]
  keepTalking.id.partner[id] -> O_Connected(id)
[]
tau{connect[id]=0;connect[partner[id]]=0;} ->
```

```
  stopTalking.id.partner[id] -> Idle(id)
[]
stopTalking.id.partner[id] -> Idle(id);

T_Alert(id) =
case {
  chan[partner[id]]!=id:
  //partner calls others before id can establish connection
  partnerBusy.id.partner[id] -> Idle(id)

  chan[partner[id]]==id:
  //partner is still calling id -> pickup
  partnerReady.id.partner[id] -> T_Pickup(id)

  default:
  // errors
  errorT_Alert.id -> ErrorState(id)
};

T_Pickup(id) =
[chan[partner[id]]==id]
  pickup.id.partner[id]{
    dev[id]=0; connect[partner[id]]=1; connect[id]=1;
  } -> T_Connected(id)
[]
[chan[partner[id]]==NIL || chan[partner[id]]!=id]
  Idle(id);

T_Connected(id) =
[connect[id]==1 && connect[partner[id]]==1]
  keepTalking.partner[id].id -> T_Connected(id)
[]
stopTalking.partner[id].id -> Idle(id)
[]
tau{connect[id]=0;connect[partner[id]]=0;} ->
  stopTalking.partner[id].id -> Idle(id);

ErrorState(id) =
error -> Stop; // an error happened
```

At the beginning of the model, we define global constants and variables. The global constant *NoOfChannels* is equal to the number of users plus 2 (i.e., 4), so that each user has its own local channel and there are two additional channels, which are left for future work on this model (e.g., one outgoing trunk and one incoming trunk). The constant *NIL* is equal to the number of users plus 1 (i.e., 3), and it designates the inactive channel. When an element of the array *chan* (a shorthand for channel) is assigned the value *NIL*, it means that the corresponding channel is inactive. The constant *INVALID_USER* is equal to the number of users (i.e., 2), and it represents the upper bound on the variable *id*, which holds an index of a user (*id* must be less than this constant).

Next, we define global variables, which we classify as the model variables and the verification variables. The model variables include the arrays *partner*, *chan*, *connect*, *dev*, *CFU*, *CFB*, *RBWF*, *lastCall*, *OCS*, *ODS*, and *TCS*, which are of size *NoOfChannels*. The verification variables are the arrays *dailNum* and *justDai*l, which are of size *NoOfUsers*. The conventions for the possible values of these variables are as follows:

The value *partner*[*i*] = *k* means that the user *i* is connecting with the user *k*, whereas the value *partner*[*i*] = *NIL* means that the user *i* is free (i.e., in the state *Idle*). The value *chan*[*i*] = *NIL* means no incoming call for the user *i*, whereas the value *chan*[*i*] = *k* means the user *k* is calling the user *i*. The value *connect*[*i*] = 1 means that the user *i* is connected to other side, whereas the value *connect*[*i*] = 0 means that it is not connected. The value *dev*[*i*] = 0 means that a device of the user *i* is busy, whereas the value *dev*[*i*] = 1 means that this device is ready.

The value *CFU*[*i*] = *NIL* means that the user *i* not subscribed to the *CFU* service, whereas the value *CFU*[*i*] = *k* means that a call to the user *i* shall be unconditionally forwarded to the user *k*. The value *CFB*[*i*] = *NIL* means that the user *i* has not subscribed to the *CFB* service, whereas the value *CFB*[*i*] = *k* means that a call to the user *i* shall be forwarded to the user *k*, if the user *i* is busy. The value *RBWF*[*i*] = 0 means that the user *i* has not subscribed to *RBWF* service, whereas the value *RBWF*[*i*] = 1 means that the user *i* shall ring back the user *lastCall*[*i*] when the user *i* becomes free. The value *lastCall*[*i*] = *NIL* means there is no last call for the user *i*, whereas the value *lastCall*[*i*] = *k* means the last call to the user *i* (when the user *i* was busy) was from the user *k*.

Generally, a screen (block) list can be implemented using a hash table. For simplicity, here we use a list of size one, i.e., just one screened (blocked) number per user, which is quite sufficient for modeling and verification purposes. Thus, the arrays *OCS*, *ODS*, and *TCS*, contain these minimal one-element lists for each user. The value *OCS*[*i*] = *NIL* means that the user *i* is not subscribed to the *OCS* service, whereas the value *OCS*[*i*] = *k* means that the user *k* is screened. The conventions for *ODS*[*i*] and *TCS*[*i*] are the same as for *OCS*[*i*]. The difference between these three services is the moment when the screening takes place (i.e., in which call-processing state).

The conventions for the verification variables are as follows: The value *dialNum*[*i*] = *k* means that the user *i* dialed the number *k* (originally), whereas the value *dialNum*[*i*] = *NIL* means there is no such number. The value *justDial*[*i*] = 0 means that the user *i* did not just dial a number, whereas the value *justDial*[*i*] = 1 means the user *i* did just dial a number.

Next, we define the processes in the model. The process *System* behaves as the process *User*, which, in turn, is defined as a concurrent execution of *NoOfUser* (i.e., 2) instances of the process *Idle*. In fact, the process *Idle* models the initial state of each user. Further on in the model, we define an individual process for each possible call-processing state of the user. Besides *Idle* these processes are *Answer*, *Dialing*, *Calling*, *Callbusy*, *CallUnobtainable*, *O_Alert*, *O_Connected*, *T_Alert*, *T_Pickup*, *T_Connected*, and *ErrorState*. The single parameter of all these processes is the user identification (*id*), where *id* is an element of the set {0..*NoOfUsers*–1}, i.e., {0, 1}. In the following text, we briefly describe each process in turn.

The process *Idle* first initializes model variables according to the conventions introduced above, in particular it sets *chan*[*id*] to *NIL*, *partner*[*id*] to *NIL*, *dev*[*id*] to 1, *connect*[*id*] to 0, and *dialNum*[*id*] to *NIL*. Further on, *Idle* nondeterministically selects one of the three possible activities (by using the external choice operator []). The guard for the first activity is that there is no incoming call to the user *id* (*chan*[*id*] == *NIL*), and in this case, *Idle* initiates the outgoing call (by setting *dev*[*id*] to 0), and transforms into the process *Dialing*. The guard for the second activity is that there is an incoming call to the user *id*, and, in this case, *Idle* accepts this incoming call (by setting *partner*[*id*] to *chan*[*id*] and *dev*[*id*] to 0), and transforms into the process *Alert*. The guard for the third activity is that the user id is subscribed to RBWF service and that there was an incoming call to the user *id* while it was busy (*RBWF*[*id*] == 1 && *lastCall*[*id*] != *NIL*), and, in this case, *Idle* checks whether the initiator of that incoming call is in the ODS screen list. If that initiator is not in the ODS list (*lastCall*[*id*] != *ODS*[*id*]), *Idle* initiates the ring back (by setting *partner*[*id*] to *lastCall*[*id*] and *lastCall*[*id*] to *NIL*) and transforms into the process *Calling*; otherwise it ignores this situation and continues to behave as the same process *Idle*.

The process *Answer* performs one of three possible cases. If the calling user (*partner*[*id*]) is in the TCS list of the user *id* (*partner*[*id*] == *TCS*[*id*]), *Answer* ignores this incoming call, and if the element *chan*[*partner*[*id*]] is set to *id*, it resets it to *NIL*, and ultimately transforms into the process *Idle*. Otherwise, if the calling user is still waiting for the user *id* to answer (*chan*[*partner*[*id*]] == *id*), *Answer* transforms into the process *T_Alert*. Otherwise (in the third case), *Answer* transforms into the process *Idle*.

The process *Dialing* nondeterministically selects one of the two possible activities. In the first activity, *Dialing* stops the outgoing call of the user *id* and transforms into the process *Idle* (this activity corresponds to the case when the user *id*, for some reason, quits the call). In the second activity, *Dialing* nondeterministically selects the called user (the variable *callId*). If this user is in the ODS list of the user *id* (*callId* == *ODS*[*id*]), *Dialing* forbids the call and transforms it to the process *Idle*. Otherwise (if the call is not screened), *Dialing* sets *partner*[*id*] to *callId* and *dialNum*[*id*] to *callId*, and transforms into the process *Calling*.

The process *Calling* first checks whether the called user (*partner*[*id*]) is in the ODS list of the user *id*, and if it is in this list, then it forbids the call and transforms into the process *Idle*. Otherwise (if the call is not screened), *Calling* performs one of the seven possible cases. The first case is when the user *id* calls itself, then *Calling* ignores the call and transforms into the process *CallBusy*. The second case is when the called user is invalid, then *Calling* ignores the call and transforms into the process *CallUnobtainable*. The third case is when the called user is subscribed to CFU service, then *Calling* sets *partner*[*id*] to *CFU*[*partner*[*id*]] and continues to behave as the process *Calling*. The fourth case is when the channel of the called user is free, then *Calling* sets *chan*[*partner*[*id*]] to *id* and *chan*[*id*] to *partner*[*id*], and transforms into the

process *O_Alert*. The fifth case is when the called user is busy and is not subscribed to CFB service, then if the called user is subscribed to RBWF service, *Calling* sets *lastCall*[*partner*[*id*]] to *id* and transforms into the process *CallBusy*, else (if the called user is not subscribed to RBWF service), *Calling* just transforms into the process *CallBusy*. The sixth case is when the called user is busy and it is subscribed to CFB service, then *Calling* sets *partner*[*id*] to *CFB*[*partner*[*id*]] and continues to behave as the process *Calling*. The seventh case is the default case (none of the previous cases, i.e., some error occurred), then *Calling* transforms into the process *ErrorState*.

The process *CallBusy* notifies the user *id* that called user is busy (by the event *soundBusy.id*) and transforms into the process *Idle*. Similarly, the process *CallUnobtainable* notifies the user *id* that the called user is invalid (by the event *soundInvalid.id*) and transforms into the process *Idle*.

The process *O_Alert* nondeterministically selects one of the four possible activities. The guard for the first activity is that the user *id* was still calling the same partner (*chan*[*id*] == *partner*[*id*]) and that partner answered the call (*connect*[*id*] == 1), and in this case *O_Alert* connects the call (by the event *callConnect.id.partner*[*id*]) and transforms into the process *O_Connected*. The guard for the second activity is that the user *id* was still calling the same partner (*chan*[*id*] == *partner*[*id*]), and that the partner did not answer the call (*connect*[*id*] == 0); in this case *O_Alert* quits the call (by the event *ringOut.id*), sets *chan*[*partner*[*id*]] to *NIL* and *chan*[*id*] to *NIL*, and transforms into the process *Idle*. The guard for the third activity is that the user *id* quit the call (*chan*[*id*] != *partner*[*id*]), and in this case *O_Alert* indicates that the call was stopped and transformed into the process *Idle*. The guard for the fourth case is that error occurred (*connect*[*id*] is neither 0 nor 1), and in this case *O_Alert* indicates that an error occurred and transformed the call into the process *ErrorState*.

The process *O_Connected* nondeterministically selects one of the three possible activities. The guard for the first activity is that both calling and called users are still connected, and in this case *O_Connected* indicates that the conversation phase is ongoing (*keepTalking.id.partner*[*id*]), and continues to behave as the process *O_Connected*. In the second activity, *O_Connected* disconnects both users (by setting *connect*[*id*] to 0 and *connect*[*partner*[*id*]] to 0), indicates the end of the conversation phase (*stopTalking.id.partner*[*id*]) and transforms into the process *Idle*—this activity corresponds to the case when the calling user *id* ends the call first. The guard for the third activity is at the end of the conversation phase (*stopTalking.id.partner*[*id*]), which has been indicated by the called user, and, in this case, *O_Connected* transforms into the process *Idle*.

The process *T_Alert* performs one of three possible cases. The first case is when the called user is busy, then *T_Alert* indicates that partner is busy (*partnerBusy.id.partner*[*id*]) and transforms into the process *Idle*. The second case is when the user *id* is still calling the same *partner*[*id*] and the called user is not busy, then *T_Alert* indicates that partner is ready (*partnerReady.id.partner*[*id*]) and transforms into the process *T_Pickup*. The third case is the default case

when some error occurs, then *T_Alert* indicates the error and transforms into the process *ErrorState*.

The process *T_Pickup* nondeterministically selects one of the two possible activities. The guard for the first activity is that the user *id* is still calling the same *partner*[*id*], and in this case *T_Pickup* indicates the partner's answer, sets *dev*[*id*] to 0, *connect*[*partner*[*id*]] to 1, and *connect*[*id*] to 1, and transforms into the process *T_Connected*. The guard for the second activity is that either no incoming call is present at the called side (*chan*[*partner*[*id*]] == *NIL*) or that the incoming call at the called side is not from the user *id* (*chan*[*partner*[*id*]] != *id*), and, in this case, *T_Pickup* transforms into the process *Idle*.

The process *T_Connected* nondeterministically selects one of the three possible activities. The guard for the first activity is that both users are still connected, then *T_Connected* indicates that the conversation phase is still ongoing (*keepTalking.partner*[*id*].*id*), and continues to behave as the process *T_Connected*. The guard for the second activity is the calling user *id* has disconnected the call (*stopTalking.partner*[*id*].*id*), then *T_Connected* transforms into the process *Idle*. The third activity is when the called user decides to disconnect the call, then *T_Connected* sets *connect*[*id*] to 0 and *connect*[*partner*[*id*]] to 0, indicates the end of the conversation phase (*stopTalking.partner*[*id*].*id*), and transforms into the process *Idle*.

The process *ErrorSate* just indicates that an error occurred and stops execution by transforming into the process *Stop*. Next, we define various system properties.

Property no. 1 states that a connection between two users is possible. We use the event *pickup.1.0* (user 1 picks up the phone dialed by user 0) to check this property. The assertion below claims that user 1 will never pick up the phone dialed by user 0. PAT finds this assertion to be invalid and produces a lengthy witness trace, demonstrating that user 1 will pick up the phone dialed by user 0, i.e., that connection between 0 and 1 is possible.

```
#assert System |= [](!pickup.1.0);
```

Property no. 2 states that if a user dials itself, then it will receive the engaged tone before it returns to the idle state. We use the events *dial.0.0* (the user 0 dials themselves), *soundBusy*.0 (user 0 hears a busy tone, i.e., is engaged), and *idle.0* (user 0 goes back to the *Idle* state) to check this property. The assertion below claims that if user 0 dials itself, then they must hear an engaged tone before returning to idle. As expected, the PAT finds this assertion to be valid.

```
#assert System |= []( dial.0.0 -> (soundBusy.0 R (!idle.0) ) ) ;
```

Property no. 3 states that either a busy tone or a ringing tone will directly follow calling. We use the events *makeCall.0* (user 0 makes an outgoing call), *soundBusy.0* (user 0 hears busy tone), *ringPartner.0* (user 0 waits for a partner to answer the call, thus hearing the ringing tone), and *soundInvalid.0* (user 0 dials an invalid number) to check this property. The assertion below claims

that if user 0 makes a call, it must hear either a busy or ringing tone before any next major event (*idle.0*, *callConnect.0*, *ringOut.0*). Note that the attribute directly in the property specification is not equal to X (next) in LTL logic, because there are other events in between. Thus, we encode the attribute directly as before any next major event. As expected, the PAT finds this assertion to be valid.

```
#assert System |= [](makeCall.0 -> (soundBusy.0 || ringPartner.0 ||
soundInvalid.0) R (!idle.0 && !callConnect.0 && !ringOut.0) );
```

Property no. 4 states that the dialed number is the same as the number of the connection attempt. We use the condition *dialed01* (user 0 has dialed user 1) and the event *doCall.0.1* (user 0 attempts to call user 1) to check this property. The assertion below claims that in all traces, if user 0 attempts to call user 1, then user 0 must have dialed user 1. As expected, the PAT finds this assertion to be valid.

```
#define dialed01 (dialNum[0]==1);
#assert System |= []( doCall.0.1 -> dialed01 );
```

Property no. 5 states that if the user dials a busy number, then either the busy line is cleared before a call is attempted, or the user will hear the engaged (busy) tone before returning to the idle state.

We use the event *dial.0.1* (user 0 dials user 1) and the condition *rcvReady* (user 1 is ready to receive a call), as well as the events *makeCall.0* (user 0 starts a call), *soundBusy.0* (user 0 hears the busy tone), and *idle.0* (user 0 goes back to *Idle* state) to check this property. The assertion below claims that if user 0 dials user 1, then either user 1 is ready to receive a call before user 0 starts making a call, or user 0 will hear busy signal before it goes back to *Idle* state. As expected, the PAT finds this assertion to be valid.

```
#define rcvReady (chan[partner[0]] == NIL);
#assert System |= [] ( dial.0.1 -> ( (rcvReady R (!makeCall.0) )
  || (soundBusy.0 R (!idle.0) ) ) );
```

Property no. 6 states that a user cannot make a call without having just dialed a number. Recall that the flag *justDial*[*id*] is set to 1 immediately after event *dial.id.**, and is cleared just after the next *makeCall.id* event. We use the conditions *justDialed0* (user 0 just dialed a number) and *justDialed1* (user 1 just dialed a number), as well as the events *makeCall.0* and *makeCall.1* to check this property. The assertion below claims that if user 0 starts making a call, then the event *dial.0.** has just happened, as indicated by the condition *justDialed0*, and the same holds for the user 1. As expected, the PAT finds this assertion to be valid.

```
#define justDialed0 justDial[0]==1;
#define justDialed1 justDial[1]==1;
#assert System |= []( (makeCall.0 -> justDialed0) &&
  (makeCall.1 -> justDialed1) );
```

Property no. 7 describes the CFU (Call Forward Unconditional) service. To verify this property, we initialize *systemCFU* by setting *CFU*[1]=2 and perform verification on that system.

The first assertion below claims that if *CFU*[1]==2, then in all traces, if user 0 dials user 1 (the event *dial.0.1*), then user 0 will call user 2 (the event *docall.0.2*) before user 0 can go back to the state *Idle* (the event *idle.0*), see the sequence diagram in the Figure 5.11. The second assertion below claims that in all the traces, user 0 will not connect to user 1. As expected, the PAT finds both assertions to be valid.

```
SystemCFU = initCFU{CFU[1]=2;} -> System();
#assert SystemCFU |= [] (dial.0.1 -> (doCall.0.2 R (!idle.0)) );
#assert SystemCFU |= ([] !callConnect.0.1);
```

Property no. 8 describes the CFB (Call Forward when Busy) service. To verify this property, we initialize a system with *CFB*[1]=2. The assertion below claims that in all the traces, if user 0 dials user 1 (the event *dial.0.1*) and user 1 is busy (the condition *Busy1*) then user 0 will not go back to the state idle before it has dialed user 2. Since the system is symmetric, similar assertions hold for other users. As expected, the PAT finds this assertion to be valid.

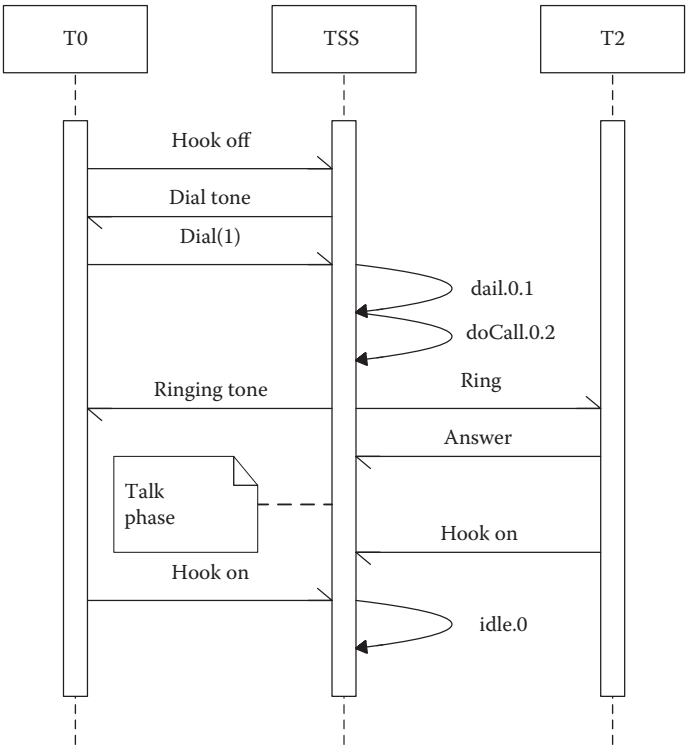


FIGURE 5.11
Sequence diagram for property no. 7.

```
SystemCFB = initCFB{CFB[1]=2;} -> System();
#define Busy1 chan[1]!=NIL && chan[1]!=0;
#assert SystemCFB |= [](  (dial.0.1 && Busy1)->
  (doCall.0.2 R (!idle.0)) );
```

Property no. 9 describes the OCS (Originating Call Screening) service. To verify this property, we initialize a system with *OCS*[0]=1. The assertion below claims that, in such a system, the event *doCall.0.1* (user 0 calls user 1) will never happen, see the sequence diagram in the Figure 5.12. Since the system is symmetric, similar assertions hold for other users. As expected, the PAT finds this assertion to be valid.

```
SystemOCS = initOCS{OCS[0]=1;} -> System();
#assert SystemOCS |= []!doCall.0.1;
```

Property no. 10 describes the ODS (Originating Dial Screening) service. To verify this property, we initialize a system with *ODS*[0]=1. The assertion below claims that, in such a system, the event *dial.0.1* (user 0 dials user 1) will never happen. As expected, the PAT finds this assertion to be valid.

```
SystemODS = initODS{ODS[0]=1;} -> System();
#assert SystemODS |= []!dial.0.1;
```

Property no. 11 describes the TCS (Terminating Call Screening) service. To verify this property, we initialize a system with *TCS*[0]=1. The assertion below claims that in such a system, the event *t_alert.0.1* (user 0 responses to a call

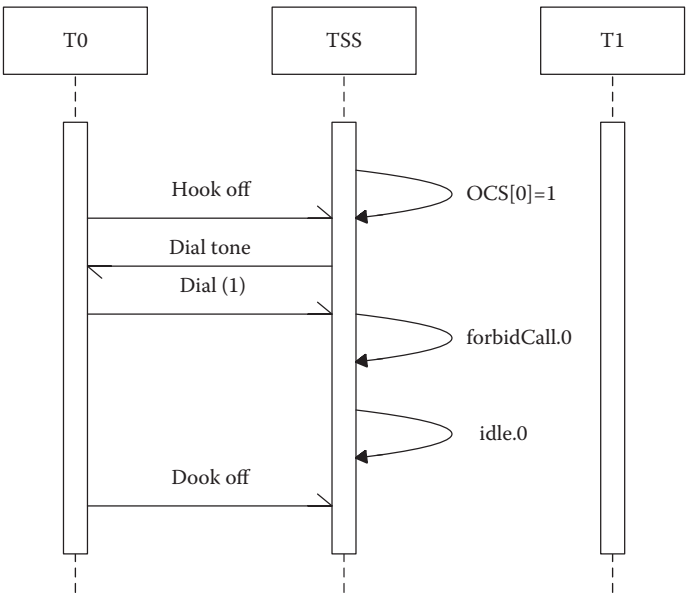


FIGURE 5.12
Sequence diagram for property no. 9.

from user 1) will never happen. Since the system is symmetric, similar assertions hold for other users. As expected, the PAT finds this assertion to be valid.

```
SystemTCS = initTCS{TCS[0]=1;} -> System();
#assert SystemTCS |= []!t_alert.0.1;
```

Property no. 12 describes the RBWF (Ring Back When Free) service. To verify this property, we initialize a system with *RBWF*[1]=1. Next, we define the supplementary condition *dial10*, meaning that user 1 must have dialed user 0. The first assertion below claims that, in such a system, the event *ringback.1.0* (user 1 rings back user 0) will never happen. As expected, the PAT finds that this assertion is not valid, and produces a lengthy counterexample. The second assertion below claims that whenever *dial.0.1* happens, if user 1 calls user 0, then user 1 must have dialed user 0 (the condition *dial10*). As expected, the PAT finds that this assertion is not valid, and produces a lengthy counterexample. The second assertion is invalid because user 1 may make a simple call to user 0 (the event *dial.0.1*) and not because user 1 is using the ring back service (the event *ringback*.1.0).

```
SystemRBWF = initRBWF{RBWF[1]=1;} -> System();
#define dialed10 dialNum[1]==0;
#assert SystemRBWF |= [] !ringback.1.0;
#assert SystemRBWF |= [](dial.0.1 -> [](callConnect.1.0 -> dialed10));
```

5.4 Statistical Usage Testing

Statistical usage testing, also referred to as *statistical testing* or *behavioral testing*, is the main industry standard for quality assessment of embedded systems today. As its name suggests, the goal of statistical usage testing is to test the product under conditions that it is expected to face in its real exploitation. The description of these conditions is given with a set of the product's operational profiles. Two key ideas are behind the concept of statistical usage testing. The first addresses the focus of testing, whereas the second addresses the quality of the final product.

We start with the genesis of the first of these two ideas. That any nontrivial product requires a vast amount of test cases for its verification should be obvious by now. The order of this amount can very easily go up to hundreds of thousands of test cases or more. Because some of the product's working modes (also referred to as states) are more frequently used than others, selecting a number of associated test cases accordingly makes sense, especially if we want to limit the size of the test suite.

This reasoning led to the concept of the operational profile. Remember that the motivation for its introduction was to respect the usage frequencies of individual operational states. Actually, because product state transitions

are triggered by corresponding events (signals, messages), the state usage frequencies are equal to the frequencies of these events. Furthermore, if we want to make our considerations independent of the total number of usages (tests), introducing the probabilities of events is convenient. (In this context, we define probability as the number of real occurrences of the event divided by the total number of its possible occurrences.)

Mathematically, the operational profile is a Markov process. It can be modeled as a special kind of graph whose vertices are product states, and whose arcs are state transitions triggered by the corresponding events of the given probability. The operational profile is essentially an FSM with given probabilities of its state transitions. Of course, the sum of probabilities of all outgoing state transitions for a single state must be equal to 1 (100%).

The second idea behind the concept of statistical usage testing is to use the product's reliability as the main measure of its quality. The genesis of this idea is that the traditional software engineering measures of product quality are the *number of remaining bugs* and the *test coverage* of the implementation under test that was achieved through its testing. However, achieving good results with respect to these two measures is not sufficient for assuring the high quality of the product.

For example, consider the following paradox: Imagine a software product that has a single bug that causes a system crash every time the software is started. Although the product has the excellent value of the metric *number of remaining bugs* (only 1 bug remaining), it is completely unreliable and therefore practically unusable. In real life, we are not interested in how good the product is with respect to the number of remaining bugs and test coverage. Rather, we are primarily interested in its reliability.

Of course, we cannot measure the product reliability directly, but we can estimate this from the number of test cases that it has successfully passed. More precisely, in real engineering practice we have the opposite problem. We want to calculate the number of test cases needed for the desired product reliability, and for the given level of risk we are ready to accept. We can do this by solving the following equation:

$$B = R^N$$

where

- B is an upper bound on the probability that the model assertions are erroneous.
- R is a lower bound on the estimate of product reliability.
- N is the number of random test cases that the product must successfully pass.

For example, achieving even moderate reliability of $R = 0.999$ with $B = 0.007$ would require the successful pass of $N = 5{,}000$ random test cases. Similarly,

achieving $R = 0.9999$ with $B = 0.007$ requires $N = 50{,}000$ random test cases, and achieving $R = 0.99999$ with $B = 0.007$ requires $N = 500{,}000$ random test cases. Alternatively, we can run a smaller number of test cases on more product samples in parallel. For example, instead of running $N = 500{,}000$ random test cases on a single sample, we can run $N = 50{,}000$ random test cases on 10 product samples simultaneously.

By considering these examples, we can deduce two conclusions. The first is that conducting statistical usage testing of the final product may require a significant amount of time. The order of magnitude of this amount is calendar weeks or even months, depending on the characteristics of the concrete product. The second conclusion is that we definitely need tools that automatically generate and execute test suites of that size. We simply cannot do this by hand.

An example of the automated working environment for generating statistical test suites is described by Popovic and Velikic (2005). This working environment consists of two parts, namely, the front-end and the back-end (Figure 5.13). The front-end is the Generic Modeling Environment (GME) developed at the Institute for Software Integrated Systems at Vanderbilt University. GME is a configurable toolkit for creating domain-specific modeling and program synthesis environments.

Generally, we configure GME by creating metamodels that specify the *modeling language*, and therefore the *modeling paradigm*, of the application domain. Once we create a metamodel, we must interpret and register it by GME to create a new working environment for making domain-specific models. We normally use such working environments for building domain-specific models and for storing them in a model database. The domain-specific models are essentially graphs, and we render them by dragging and dropping the graphical symbols on the working sheet that is maintained by the GME graphical user interface (GUI). The symbols in GME have their attributes, preferences, and properties.

The particular metamodel that specifies the language (and the paradigm) for modeling operational profiles is represented with the metaclass *OperationalProfile* in Figure 5.13. Each concrete operational profile model (represented with the class *OpProfile* in Figure 5.13) is created by using the operational profile modeling paradigm (the class *OpProfile* is derived from the class *OperationalProfile*). Creating operational profile models by using this paradigm is quite easy.

The modeling language for rendering operational profile models has a single symbol, *State*. This symbol has a single attribute, which is the name of the state. Normally, we just drag and drop the state symbol icon to the working sheet, click on the name field, and type in its name. Each of the state symbols we place on the working sheet represents a single working state (mode) of the product that we want to test.

Rendering state transitions requires a little more work. To render a state transition, we select a connecting tool (symbolized by the operator "+"), click on the source state, and click on the destination state. When the state

transition is in place, we enter the particular data for its attributes. A state transition has the following three attributes:

- *EventClass* specifies the class of events that trigger the state transition.
- *Output* specifies the expected output of the state transition.
- *Probability* specifies the probability of the state transition (in percent).

The most frequently used format of the attribute *EventClass* definition is as follows:

```
E(a,b,c...);->a := A1/A2/...; b := B1/B2/...; c := C1/C2/...
```

The event class definition above consists of two parts. The first one is on the left-hand side of the substring "->" and is referred to as the event class. The event class *E(a,b,c…);* is a string with an arbitrary number of parameters (substrings), labeled here as *a*, *b*, *c*, and so on. The second part of the definition is on the right-hand side of the substring "->". It provides definitions of possible replacements (which are also strings) for each event class parameter. As indicated above, the parameter *a* may be replaced with the string A_1 or A_2 and so on.

A particular event (also referred to as the *constant event*) is an event class without parameters. We may also think about it as the event class with a single member. Particular events are generated from the event class by substituting each event class parameter with the randomly selected replacement from the list of possible replacements. All replacements have equal selection probabilities. Examples of particular events for the event class definition given above are $E(A_1,B_1,C_1\ldots)$, $E(A_1,B_1,C_2\ldots)$, $E(A_1, B_2, C_1\ldots)$, $E(A_2, B_1, C_1\ldots)$, and so on.

The event class format shown above is feasible as far as the number of the possible values of event class parameters is relatively small. But when the number of the possible values is large, writing them explicitly becomes impractical, if not impossible. For example, consider the integer parameter whose possible values are from the interval [0,10000). Writing all 10,000 of its possible values would be really annoying. To make it easier for the user, the working environment supports the following two intrinsic functions:

- *randInt<i,j>* randomly selects an integer number from the interval [*i,j*).
- *randFloat<x,y>* randomly selects a float number from the interval [*x,y*).

When we place and name all state symbols, interconnect them with state transitions, and enter the data for attributes of all state transitions, the operational profile model is finished, and we can store it in a file (or a database).

This is exactly the main purpose of the working environment front-end (Figure 5.13). Of course, later we may modify the model by adding or deleting states or state transitions, as well as by changing the data for attributes of state transitions, and store it again. All these manipulations are supported by the GME's GUI.

The working environment back-end consists of two parts. The first is the operational profile model interpreter (represented by the class *ModelInterpreter* in Figure 5.13), which is registered to GME. The second part of the back-end is a separate program written in Java, which is named Generic Test Case Generator (GTCG). The main task of the model interpreter is to transform the operational profile model to the operational profile specification, a simple text file of the well-defined format (represented with the class *OpProfileSpec* in Figure 5.13). Alternately, the main task of GTCG is to automatically generate the test suite to be used for statistical usage testing and the corresponding statistical report (represented with the classes *TestSuite* and *Statistics* in Figure 5.13).

The operational profile model interpreter is a Java package that is registered to GME with the program JavaCompRegister. The package comprises the following three classes:

- *OPBONComponent*: the interface between GME and the model interpreter
- *OPState*: the state interpreter
- *OPTransition*: the state transition interpreter

The model interpreter behaves similarly to traditional plug-in components of GUIs. We activate it by a click on the corresponding model interpreter icon. As the result of this activation, GME calls the model interpreter interface function *invokeEx*, which, in turn, creates temporary container objects for state names, event classes, state transition probabilities, event class definitions, and next state definitions.

Next, the model interpretation is performed by traversing the multigraph architecture of the model in focus. While visiting individual states and state transitions, GME calls the function *traverseChildren* of the classes *OPState* and *OPTransition*, respectively. These two functions effectively interpret the model by reading the data of the attributes and filling the above-mentioned container objects. At the end of the interpretation, the content of these container objects is saved into the operational profile specification file named *opspec.txt*.

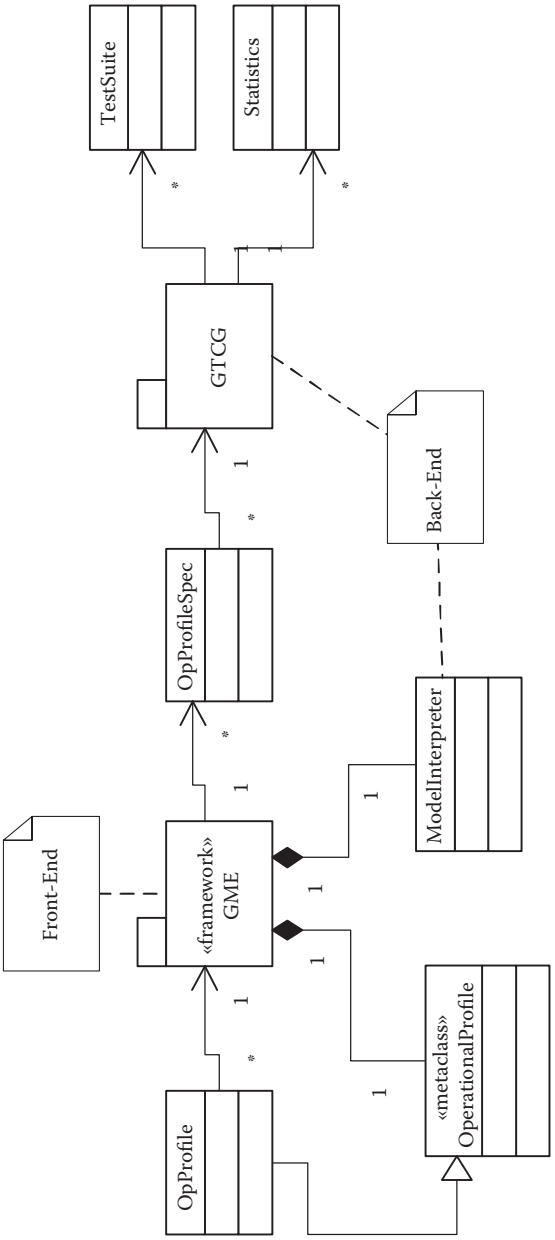


FIGURE 5.13
Working environment for generating statistical test suites.

The automatic test case generator GTCG uses the following input items:

- The operational profile data from the file *opspec.txt*.
- The initial operational profile state: Most frequently, the initial state is fixed, but sometimes it may be selectable.
- The number of test cases to be generated: This item determines the size of the test suite. As mentioned earlier, it depends on the product reliability we want to guarantee.
- The test case length, defined as the number of test steps in a test case. A test step is the particular event that is randomly selected from the given event class.

The operational profile specification file *opspec.txt* consists of the following four parts:

- Part I defines the number of states (M) and the number of event classes (N).
- Part II is a matrix of state transition probabilities. The matrix element P_{ij} defines the probability of the event class number j in the operational profile state number i.
- Part III is a matrix of event class definitions. The matrix element E_{ij} defines the event class number j in the operational profile state number i. Most frequently, E_{ij} is the same in all states ($E_{i1} = E_{i2} = \ldots E_{iM}$).
- Part IV is a matrix of next states. The matrix element T_{ij} defines the next state number (index) for the event class number j in the operational profile state number i.

GTCG provides the following two files at its output:

- *testcases.txt* contains the test suite to be used for statistical usage testing.
- *statistics.txt* contains the corresponding statistical report, which is the important measure of the generated test suite quality.

The file *testcases.txt* contains the series of test cases. Each test case starts with its number followed by the column character ':' (e.g., 0:, 1:, 2:). The next line contains the test bed setup command *TestBox.initialize()*, which essentially initializes the hardware connected to product inputs and outputs for the purpose of automatic testing. The test bed setup command is followed by the series of lines that contain particular events randomly selected from the associated event classes (the number of these lines is determined by the given test case length). The event class itself is selected

randomly from the distribution defined by the operational profile data (*opspec.txt*, Part II).

The file *statistics.txt* consists of two parts. The first part contains a series of lines, one per operational profile state. Each of these lines indicates the number of occurrences of the corresponding operational profile state (c_i), the discrepancy between the observed and expected frequency of state occurrence (d_i), and the significance level (SL_i). The significance level is actually the probability that the discrepancies as large as those observed would occur with random variation. The second part of the statistical report shows the mean value of the discrepancy and the mean value of the significance value.

A detailed explanation of the statistical measures mentioned above is outside the scope of this book but can be found elsewhere (e.g., Woit, 1994). Practically, it is enough to remember the following guides:

- A significance level greater than or equal to 20% is considered large. This result means that the test suite is of sufficient quality and we may use it for statistical usage testing.
- A significance level less than or equal to 1% is considered small. This result means that the test suite quality is poor and it should not be used for statistical usage testing.

The statistical usage testing methodology governs the usage of tools that create the working environment. This methodology subsumes the following steps:

- Make the operational profile model of the product (implementation under test).
- Interpret the model.
- Determine the desired level of reliability.
- Calculate the required size of the test suite (the number of test cases).
- Generate the test suite.
- Check the test suite quality. If the quality is not acceptable, return to the previous step.
- Execute the test suite. If all test cases successfully pass, the final verdict is *pass*. In that case, we can claim that product reliability is at least at the level of the desired reliability. If at least one test case fails, the final verdict is *fail* and the product is considered not usable, at least not at the desired level of reliability.

This methodology can be used for testing both parts of the products and their complete forms. We will illustrate such applications by the following two examples. The implementation under test in the first example is the SIP

invite client transaction. We start with modeling its operational profile in accordance with the methodology outlined above (Figure 5.14).

The operational profile shown in Figure 5.14 has five working states, namely, *Initial*, *Calling*, *Proceeding*, *Completed*, and *Terminated*. At the same time, it has nine event classes that are intentionally labeled with names that resemble the original specification (see RFC 3261, Figure 5). The definitions of the event classes (not shown in Figure 5.14) are the following:

- The event class labeled *INVITE* is defined as *INVITE* (this class has a single member).
- The event class labeled *300–699* is defined as `M3->M3:=randInt<300,700>;`
- The event class labeled *TA* is defined as TA (original RFC 3261 label: Timer A fires).
- The event class labeled *1XX* is defined as `M1->M1:=randInt<100,200>;`
- The event class labeled *TB* or *TransportERR* is defined as `E->E:=TB/TransportERR;`
- The event class labeled *2XX* is defined as `M2->M2:=randInt<200,300>;`
- The event class labeled *TD* is defined as TD (original RFC 3261 label: Timer D fires).
- The event class labeled *TransportERR* is defined as TransportERR (constant event).
- The event class labeled *End* is defined as End (added because the sum of outgoing state transition probabilities for each state must be equal to 100%).

The probabilities of individual state transitions are shown in Figure 5.14. Note that outgoing state transition probabilities add up to 100% for each state (an essential request for a Markov process). Generally, we set the state transition probabilities according to what we expect the product will face in its real exploitation. Of course, we should use statistical data available for some similar product or the previous version of the same product whenever we can.

Next, we start the model interpreter, which transforms the model into the operational profile specification file *opspec.txt*. When writing GME model interpreters, we should make no assumptions about the order in which the model is traversed. For example, assuming that individual states and state transitions are going to be visited in the same order in which they were originally entered would be a mistake because this is not going to happen. The best assumption we can make in this respect is to assume a completely random visiting order.

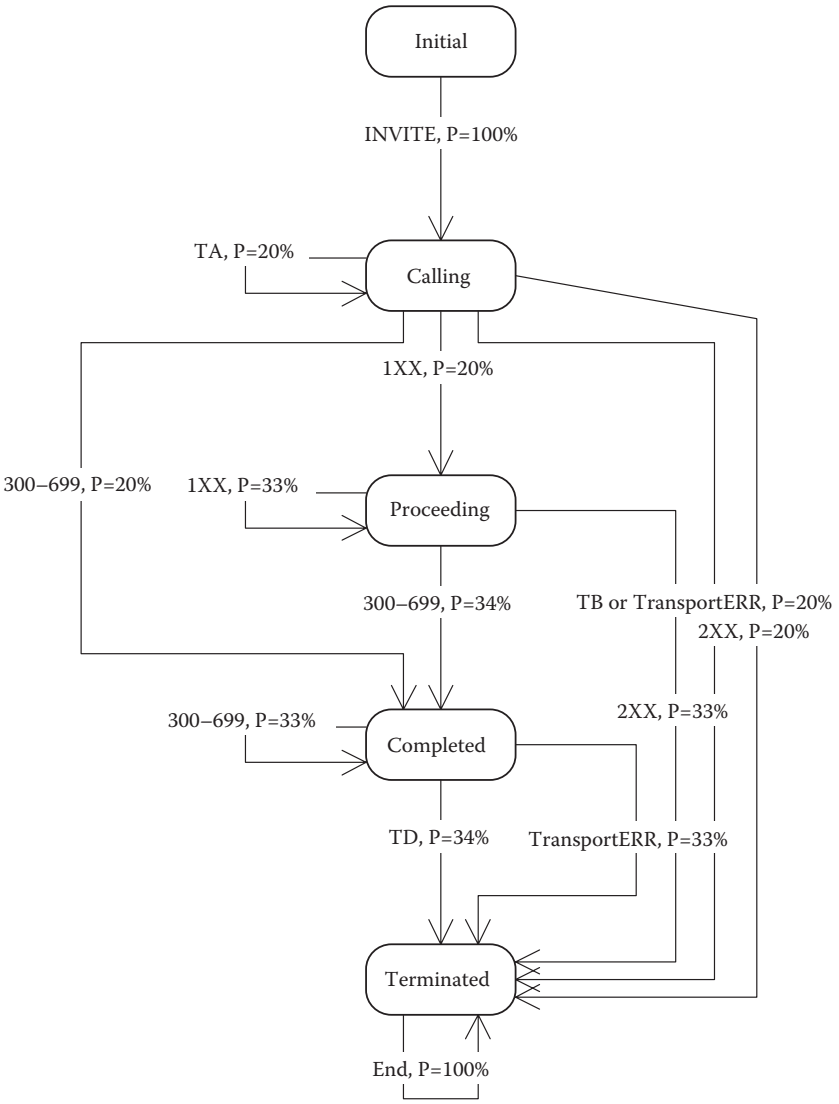


FIGURE 5.14
SIP INVITE client transaction operational profile.

Based on this assumption, the model interpreter simply assigns identifications to states and state transitions according to the order they are visited. The particular assignment of identifications to operational profile states in this example is the following:

- The state *Terminated* is assigned the identification 0.
- The state *Calling* is assigned the identification 1.

- The state *Proceeding* is assigned the identification 2.
- The state *Completed* is assigned the identification 3.
- The state *Initial* is assigned the identification 4.

The particular assignment of identifications to operational profile event classes is the following:

- The event class *E* is assigned the identification 0.
- The event class *M1* is assigned the identification 1.
- The event class *TA* is assigned the identification 2.
- The event class *INVITE* is assigned the identification 3.
- The event class *TransportERR* is assigned the identification 4.
- The event class *M2* is assigned the identification 5.
- The event class *M3* is assigned the identification 6.
- The event class *TD* is assigned the identification 7.
- The event class *End* is assigned the identification 8.

The content of the file *opspec.txt* is the following:

```
5       9

0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     1.0

0.2     0.2     0.2     0.0     0.0     0.2     0.2     0.0     0.0

0.0     0.33    0.0     0.0     0.0     0.33    0.34    0.0     0.0

0.0     0.0     0.0     0.0     0.33    0.0     0.33    0.34    0.0

0.0     0.0     0.0     1.0     0.0     0.0     0.0     0.0     0.0

null null null null null null null null End
E->E:=TB/TransportERR; M1->M1:=randInt<100,200>; TA null null
  M2->M2:=randInt<200,300>; M3->M3:=randInt<300,700>; null null
null M1->M1:=randInt<100,200>; null null null
  M2->M2:=randInt<200,300>; M3->M3:=randInt<300,700>; null null
null null null null TransportERR null M3->M3:=randInt<300,700>; TD null
null null null INVITE null null null null null

0   0   0   0   0   0   0   0   0
0   2   1   0   0   0   3   0   0
0   2   0   0   0   0   3   0   0
0   0   0   0   0   0   3   0   0
0   0   0   1   0   0   0   0   0
```

NOTE: The specifications of event classes for the states 1, 2, and 3 (*Calling*, *Proceeding*, and *Completed*) were too long to fit into a single line. Therefore, definitions of event classes for each of these states spans across two lines (the second starts at the next level of indentation).

Next, we activate GTCG with the script that specifies the starting state identification 4 (*Initial*), the number of test cases that is equal to 1,000, and the test case length that is equal to 4 (this means 4 steps, i.e., particular events, per test case). Selection of this particular test case length requires a short comment. This value is exactly the length of the shortest path across all five states, starting from the state *Initial* (path *Initial–Calling–Proceeding–Completed–Terminated*, with five states and four state transitions). Of course, other paths of length 4 are possible and will be generated.

As already mentioned, the GTCG creates two output files, *testcases.txt* and *statistics.txt*. According to the methodology outlined above, we first check the quality of the generated test suite by inspecting the file *statistics.txt*. Its content is the following:

```
Calculating statistics

i=0   ci=1104                  di=0.0                    SLi=1.0
i=1   ci=1237                  di=2.470493128536783      SLi=0.0
i=2   ci=291                   di=0.7498208280500565     SLi=0.7014229616104999
i=3   ci=368                   di=0.1864198248469353     SLi=0.910066579962014
i=4   ci=1000                  di=0.0                    SLi=1.0
Mean  d=0.6813467562867549
Mean  SL=0.7222979083145027
```

The average significance level *SL* is equal to 72% (0.72). Because this number is greater than the required 20%, we conclude that the quality of the generated test suite is sufficient, and that we can use it for statistical usage testing.

Next, we look more closely at a couple of test cases from the beginning of the file *testcases.txt* to get a better feeling of the nature of statistical test cases. The relevant comments are interleaved with the test cases:

```
0:
TestBox.initialize();
INVITE
443
TransportERR
End
```

Test case number 0: After the initial *INVITE*, GTCG randomly selects the event class labeled *300–699* and the particular event 443 from that class. This action causes the state to transition to the state *Completed* (Figure 5.14). Next, GTCG randomly selects the event *TransportERR*, thus causing the state to transition to the state *Terminated*. *End* is the only possible event in that state.

```
1:
TestBox.initialize();
INVITE
TA
586
TD
```

Test case number 1: After the initial *INVITE*, GTCG randomly selects the event class *TA* (Timer A fires). The current state remains in the state *Calling* (Figure 5.14). Next, GTCG randomly selects the event *586*, thus causing the state transition to the state *Completed*. Finally, GTCG randomly selects the event *TD* (Timer D fires), which causes the state transition to the state *Terminated*.

```
2:
TestBox.initialize();
INVITE
190
267
End
```

Test case number 2: After the initial *INVITE*, GTCG randomly selects the event class *1XX* and the particular event *190*. This causes the state transition to the state *Proceeding* (Figure 5.14). Next, GTCG randomly selects the event *267*, thus causing the state transition to the state *Terminated*. The next event must be the event *End*.

```
3:
TestBox.initialize();
INVITE
494
TD
End
```

Test case number 3: After the initial *INVITE*, GTCG randomly selects the event class *300–699* and the particular event *494*. This causes the state transition to the state *Completed* (Figure 5.14). Next, GTCG randomly selects the event *TD*, thus causing the state transition to the state *Terminated*. The next event must be the event *End*.

In the short descriptions of the generated test cases given above, we used the construct, "GTCG randomly selects the event class X and the particular event Y," for brevity. One should remember that the selection of the event class is always in accordance with the given operational profile probability distribution, whereas the selection of the particular event from the given class is really random.

The previous example shows how we can use statistical usage testing for testing a part of the product. As already mentioned, we can employ statistical usage testing for testing whole products, too. The next example shows such an application—statistical usage testing of the simple SIP softphone.

The operational profile of the SIP softphone is shown in Figure 5.15. It has 8 states and 13 event classes. The states are *Connecting*, *Terminating*, *Disconnecting*, *Connected*, *Calling*, *Initial*, *Proceeding*, and *Ringing* (listed here in the ascending order of their identification). The event classes are *RELEASE*, *200*, *ACK*, *180*, *ERR*, *END*, *ANSWER*, *100*, *INVITE*, *SETUP*, *BYE*, *TH*, and *TB* (also listed in the ascending order of their identification).

All event classes have just one member, and their definition is equal to the label shown in Figure 5.15, with the exception of the event class that is labeled *ERR*, which is defined as follows:

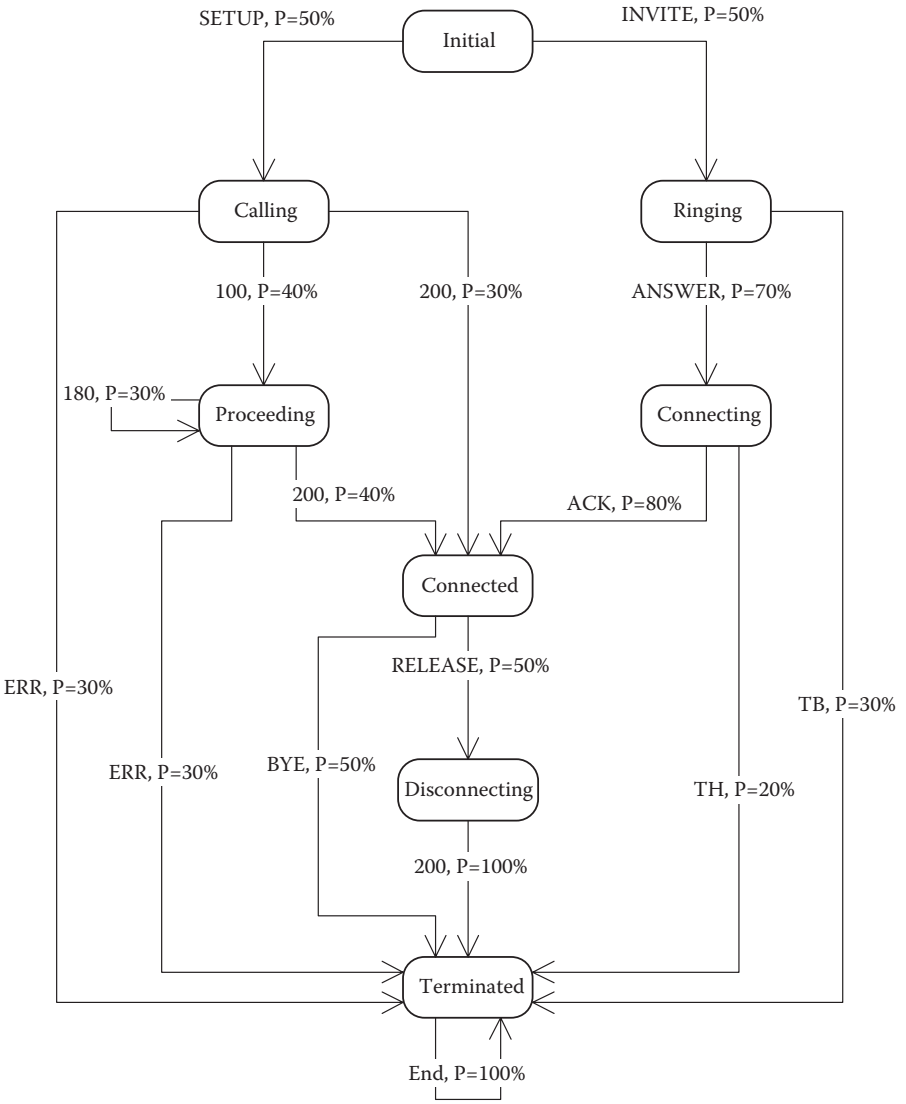


FIGURE 5.15
SIP softphone operational profile.

```
M3->M3:=randInt<300,381>/randInt<400,494>/randInt<500,514>/randInt<600,607>;
```

This definition is a good example of how we can specify a random value that may be selected from more disjointed intervals of values. Next, we generate 1,000 test cases with five test steps each. The content of the file *statistics.txt* is the following:

```
Calculating statistics

i=0   ci=360                  di=0.625                  SLi=0.4686783191616166
i=1   ci=1564                 di=0.0                    SLi=1.0
i=2   ci=244                  di=0.0                    SLi=1.0
i=3   ci=546                  di=1.6483516483516483     SLi=0.21453651135488572
i=4   ci=496                  di=0.5843413978494628     SLi=0.7503695231083775
i=5   ci=1000                 di=0.064                  SLi=0.8248262531456066
i=6   ci=286                  di=3.0879953379953404     SLi=0.21451818049555796
i=7   ci=504                  di=0.4897959183673477     SLi=0.4966702889206116
Mean  d=0.8124355378204748
Mean  SL=0.6211998845233321
```

Because the average significance level is 62% (greater than 20%), we can conclude that the test suite quality is acceptable. A couple of typical test cases are taken from the file *testcases.txt* and are shown here without comment (the reader should study them for their own exercise):

```
15:
TestBox.initialize();
SETUP
100
180
200
BYE

16:
TestBox.initialize();
INVITE
ANSWER
ACK
BYE
END

17:
TestBox.initialize();
SETUP
100
200
BYE
END

18:
TestBox.initialize();
INVITE
ANSWER
ACK
RELEASE
200
```

5.5 Examples

This section includes two examples and two related problems. The first example demonstrates unit testing of the FSM Library–based implementations. The second example illustrates integration testing of FSM Library–based products.

5.5.1 Example 1

This example demonstrates unit testing of the SIP invite client transaction implementation, which is described in Section 4.5.2 (Example 2). The SIP invite client transaction implementation is based on the requirements and analysis made in Section 2.3.3 (Figure 2.16) and the design presented in Section 3.10.5 (Example 5).

Because the implementation under test (SIP invite client transaction) is implemented in C++, we use CppUnit implementation of the unit testing framework, introduced in Section 5.1. In this simple example, we will construct just one test case to keep it short enough. Also, we will skip some SIP message-specific message handling, which is really not essential for this example.

We start this example by constructing two classes: *ExampleTestCase* and *ExampleMessageFactory*. The former is the tester class, which comprises one sample test case, whereas the latter is the supplementary class, which provides the functions for message management. The content of the class *ExampleTestCase* declaration file, named *ExampleTestCase.h,* is the following:

```
#ifndef CPP_UNIT_EXAMPLETESTCASE_H
#define CPP_UNIT_EXAMPLETESTCASE_H
// CppUnit helper macros
#include <cppunit/extensions/HelperMacros.h>
// Problem specific headers
#include "../kernel/fsmsystem.h"
#include "../kernel/logfile.h"
#include "../NewSIP/InvClientTE.h"
#include "ExampleMessageFactory.h"
/*
 * A sample test case
 *
 */
class ExampleTestCase : public CPPUNIT_NS::TestFixture {
 CPPUNIT_TEST_SUITE(ExampleTestCase);
 CPPUNIT_TEST(example);
 CPPUNIT_TEST_SUITE_END();

protected:
 FSMSystemWithTCP *pSys;
 LogFile *lf;
 InviteClientTE* pInviteCltTE[NUMBER_OF_TES];
 ExampleMessageFactory* pEMF;
 uint8 *msg;
 uint16 msgcode;
public:
 void setUp();
protected:
 void example();
};
#endif
```

The declaration file above includes the CppUnit helper macros header file (*HelperMacros.h*) and the problem-specific header files (*fsmsystem.h*, *logfile.h*, *InvClientTE.h*, and *ExampleMessageFactory.h*). The class *ExampleTestCase* is derived from the class that is defined by the macro instruction

CPPUNIT_NS::TestFixture. The definition of the test suite starts with the macro instruction *CPPUNIT_TEST_SUITE()* and ends with the macro instruction *CPPUNIT_TEST_SUITE_END()*. The parameter of the former macro instruction is the name of the test suite (*ExampleTestCase*, in this example).

Generally, we use the macro instruction *CPPUNIT_TEST()* to define individual test cases inside the body of test suite definition. The parameter of this macro instruction is the name of the test case function that is defined within the tester class and that we want to add to the test suite. In this particular example, we add a single test case function, named *example()*, with a single macro instruction, *CPPUNIT_TEST()*, whose real parameter is the string "example".

Next, we define the test case fixture. In this example, it comprises the following:

- The pointer to the instance of the class *FSMSystemWithTCP* (see Section 6.8.9)
- The pointer to the instance of the class LogFile (which is the interface to the log file)
- The array of pointers to the instances of the class InviteClientTE (which is actually the implementation under test)
- The pointer to the instance of the class ExampleMessageFactory (which is the supplementary tester class)
- The pointer to the message
- The code of the message

At the end of this file we declare the function *setUp()* and the test case function *example()*. The content of the class *ExampleTestCase* definition file, named *ExampleTestCase.cpp*, is as follows:

```
#include "ExampleTestCase.h"
#include "../kernel/fsmsystem.h"
#include "../kernel/logfile.h"
#include "../NewSIP/InvClientTE.h"
#include "ExampleMessageFactory.h"

CPPUNIT_TES_SUITE_REGISTRATION(ExampleTestCase);
void ExampleTestCase::setUp() {
  pSys = new FSMSystemWithTCP(11,11);
  pEMF = new ExampleMessageFactory();
  for (int i = 0; i < NUMBER_OF_TES; i++) {
    pInviteCltTE[i] = new InviteClientTE();
  }

  uint8 buffClassNo = 4;
  uint32 buffsCount[4] = {50, 50, 50, 50};
  uint32 buffsLength[4] = {1025, 1025, 1025, 1025};
  pSys->InitKernel(buffClassNo, buffsCount, buffsLength, 1);
```

```
  lf = new LogFile("log.log", "log.ini");
  LogAutomateNew::SetLogInterface(lf);

  pSys->Add(pInviteCltTE[0], InviteClientTE_FSM, 10, true);
  for (i = 1; i < NUMBER_OF_TES; i++){
    pSys->Add(pInviteCltTE[i], InviteClientTE_FSM);
  }
}

void ExampleTestCase::example() {
  msg = pEMF->MakeInviteToTALMsg();
  pInviteCltTE[0]->Process(msg);
  msgcode = pEMF->GetMsgCodeFromMBX(TLI_Test_FSM_MBX);
  CPPUNIT_ASSERT_EQUAL(msgcode,(uint16)INVITE);

  msg = pEMF->Make1XXToTAL();
  pInviteCltTE[0]->Process(msg);
  msgcode = pEMF->GetMsgCodeFromMBX(UA_Disp_FSM_MBX);
  CPPUNIT_ASSERT_EQUAL(msgcode,(uint16)RESPONSE_1XX);

  msg = pEMF->Make2XXToTAL();
  pInviteCltTE[0]->Process(msg);
  msgcode = pEMF->GetMsgCodeFromMBX(UA_Disp_FSM_MBX);
  CPPUNIT_ASSERT_EQUAL(msgcode,(uint16)RESPONSE_2XX);
}
```

At the beginning of this file, we register the test suite with the macro instruction *CPPUNIT_TEST_SUITE_REGISTRATION()*. The real parameter of this macro instruction is the name of the test suite. Next, we define the function *setup()* and the test case function *example()*.

The function *setup()* starts by creating an instance of the class *FSMSystemWithTCP*, an instance of the class *ExampleMessageFactory*, and the given number (*NUMBER_OF_TES*) of instances of the implementation under test (the class *InviteClientTE*). After that, it defines the types of buffers to be used by the FSM Library kernel, initializes the kernel by calling the function *InitKernel()* (see Section 6.8.4), creates the log file by calling the function *LogFile()*, and sets the log interface by calling the function *SetLogInterface()* (see Section 6.8.105). At the end, it adds the given number (*NUMBER_OF_TES*) of instances of the implementation under test to the FSM system by calling its function *Add()* (see Section 6.8.2 and Section 6.8.3).

The function *example()* performs the test case by checking state transitions of the implementation under test in the following three steps:

- Check the state transition from the state *STATE_IDLE* (see Section 4.5.2) to the state *STATE_CALLING*, driven by the message *INVITE*.
- Check the state transition from the state *STATE_CALLING* to the state *STATE_PROCEEDING*, driven by the message *1XX*.
- Check the state transition from the state *STATE_PROCEEDING* to the state *STATE_INITIAL*, driven by the message *2XX*.

Each of these three steps consists of the following four substeps:

- Create the message (*INVITE*, *1XX*, or *2XX*).
- Send the message to the implementation under test by calling its function member *Process()* (see Section 6.8.82).
- Get the message code of the output message by calling the function member *GetMsgCodeFromMBX()* of the class *ExampleMessageFactory*. The output message is retrieved from the destination FSM Library mailbox. The destination mailbox is either the mailbox of the transport layer (TPL) or the mailbox of the transaction user (TU).
- Check the retrieved message code against the expected one (message code of the message *INVITE*, *1XX*, or *2XX*) by calling the macro *CPPUNIT_ASSERT_EQUAL()*.

The particular substeps of the first step are the following:

- Create the message *INVITE* by calling the function member *MakeInviteToTALMsg()* of the class *ExampleMessageFactory*.
- Send the message to the implementation under test.
- Get the message code of the message that is retrieved from the TPL mailbox.
- Check it against the code of the message *INVITE*.

The particular substeps of the second step are the following:

- Create the message *1XX* by calling the function member *Make1XXToTAL()* of the class *ExampleMessageFactory*.
- Send the message to the implementation under test.
- Get the message code of the message that is retrieved from the TU mailbox.
- Check the message code against the code of the message *1XX*.

The particular substeps of the third step are the following:

- Create the message *2XX* by calling the function member *Make2XXToTAL()* of the class *ExampleMessageFactory*.
- Send the message to the implementation under test.
- Get the message code of the message that is retrieved from the TU mailbox.
- Check the message code against the code of the message *2XX*.

Next, we construct the supplementary class *ExampleMessageFactory*. The content of its declaration file, named *ExampleMessageFactory.h*, is as follows:

```
#ifndef _ExampleMessageFactory_FSM_
#define _ExampleMessageFactory_FSM_
#include "../constants.h"
#include "../kernel/fsm.h"
#include "../message/message.h"
class ExampleMessageFactory : public FiniteStateMachine {
  int cseq_number;
  Message SIPMsg;
  sip_t *mes;
  stringresponseBody;
public:
  uint8* MakeInviteToTALMsg();
  uint16 GetMsgCodeFromMBX(uint8 mbx);
  uint8* Make1XXToTAL();
  uint8* Make2XXToTAL();

  // FiniteStateMachine abstract functions
  StandardMessage StandardMsgCoding;
  MessageInterface *GetMessageInterface(uint32 id);
  void SetDefaultHeader(uint8 infoCoding);
  void SetDefaultFSMData();
  void NoFreeInstances();
  void Reset();
  uint8 GetMbxId();
  uint8 GetAutomate();
  uint32 GetObject();
  void ResetData();
public:
  ExampleMessageFactory();
  ~ExampleMessageFactory();
  void Initialize();
};
#endif
```

The content of the class *ExampleMessageFactory* definition file, named *ExampleMessageFactory.cpp*, is as follows (the parts that are not essential for this example are omitted to keep the example short):

```
#include "ExampleMessageFactory.h"
#include "../parser/smsgtypes.h"
#include "../parser/smsg.h"
#define SipMessageCoding 0x00
extern char* IPString(unsigned int addr, char* buf, int len);

ExampleMessageFactory::ExampleMessageFactory() : FiniteStateMachine(16, 2, 3) {}

ExampleMessageFactory::~ExampleMessageFactory() {}

void ExampleMessageFactory::Initialize() {}

uint8* ExampleMessageFactory::MakeInviteToTALMsg(){
  char temp[10];
  char szHostName[255];
  hostent* HostData;
  uint8* recmsg;
  uint8* msg;
  ...
  PrepareNewMessage(0x00,INVITE);
```

```
  SetMsgToAutomate(InviteClientTE_FSM);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(0);
  AddParam(SIP_RAW_MESSAGE, SIPMsg.getLastMessage().length(),
         (uint8*) SIPMsg.getLastMessage().c_str());
  AddParamDWord(SIP_PARSED_MESSAGE, (unsigned long) mes);
  SendMessage(InviteClientTE_FSM_MBX);
  msg = GetMsg(InviteClientTE_FSM_MBX);
  return msg;
}
uint16 ExampleMessageFactory::GetMsgCodeFromMBX(uint8 mbx) {
  uint8* msg;
  uint16 msgCode;
  msg = GetMsg(mbx);
  msgCode = GetUint16((uint8*)(msg+MSG_CODE));
  return msgCode;
}

uint8* ExampleMessageFactory::Make1XXToTAL(){
  uint8* msg;
  ...
  PrepareNewMessage(0x00,RESPONSE_1XX_T);
  SetMsgToAutomate(TAL_Disp_FSM);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(0);
  AddParamDWord(SIP_PARSED_MESSAGE, (unsigned long) mes);
  SendMessage(InviteClientTE_FSM_MBX);
  msg = GetMsg(InviteClientTE_FSM_MBX);
  return msg;
}

uint8* ExampleMessageFactory::Make2XXToTAL(){
  uint8* msg;
  SIPMsg.makeResponse("200","OK",responseBody,0);
  PrepareNewMessage(0x00,RESPONSE_2XX_T);
  SetMsgToAutomate(TAL_Disp_FSM);
  SetMsgToGroup(INVALID_08);
         SetMsgObjectNumberTo(0);
  AddParamDWord(SIP_PARSED_MESSAGE, (unsigned long) mes);
  SendMessage(InviteClientTE_FSM_MBX);
  msg = GetMsg(InviteClientTE_FSM_MBX);
  return msg;
}
...
```

The main reason we must introduce the supplementary class *ExampleMessageFactory* is because most of the functions defined in the FSM Library API are protected, which means that they cannot be used in the tester class directly. Alternately, as defined at the moment, CppUnit does not allow us to use multiple inheritance when we are defining tester classes. Rather, a tester class may be derived only from the class that is defined by the macro instruction *CPPUNIT_NS::TestFixture*.

The source code from the file *ExampleMessageFactory.cpp* should be obvious by now. The only detail that deserves a short explanation is the method by which we create messages. We use typical snippets of code, which start with the *PrepareNewMessage()* function call and are followed with the series of *SetXX()* and *AddParamXX()* function calls. The way we end these code snippets may seem odd. First, we send a new message by

calling the function *SendMessage()* and, immediately after that, we read that message from the same destination mailbox by calling the function *GetMsg()*. Although it may seem odd, this is the most effective method of creating the complete message in the format that is expected by the function *Process()*.

Finally, we write the main module, named *Main.cpp*. This module creates the collaboration of objects necessary to automatically execute the test suite and report the results of its execution (Figure 5.16). The function *main()* performs the following steps:

- Create the event manager and the test controller.
- Add a listener that collects test results.
- Add a listener that prints dots as test cases are executed (one dot per test case).
- Add the top suite to the test runner.
- Print the test results in a compiler-compatible format.

The source code of the module *Main.cpp* follows:

```
#include <cppunit/BriefTestProgressListener.h>
#include <cppunit/CompilerOutputter.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/TestResult.h>
#include <cppunit/TestResultCollector.h>
#include <cppunit/TestRunner.h>

int main(int argc,char* argv[]) {
  CPPUNIT_NS::TestResult controller;
  CPPUNIT_NS::TestResultCollector result;
  controller.addListener(&result);
  CPPUNIT_NS::BriefTestProgressListener progress;
```

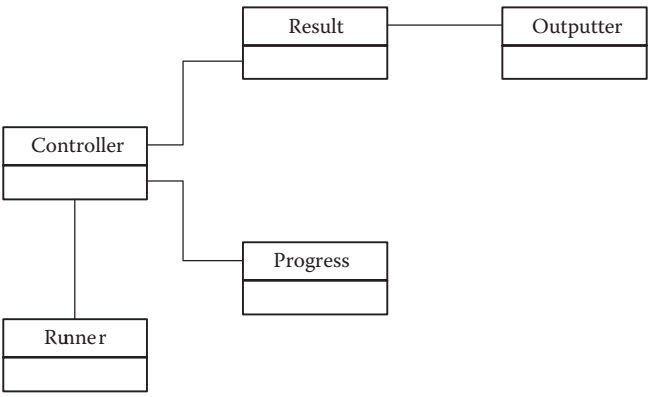


FIGURE 5.16
Collaboration of objects necessary for the automatic execution of the CppUnit test suite.

```
  controller.addListener(&progress);
  CPPUNIT_NS::TestRunner runner;
     runner.addTest(CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest());
  runner.run(controller);
  CPPUNIT_NS::CompilerOutputter    outputter(&result,std::cerr);
  outputter.write();
  return result.wasSuccessful() ? 0 : 1;
}
```

As a result of automatic test suite execution, we get the following report on the monitor:

```
ExampleTestCase::example : OK
OK(1)
Press any key to continue...
```

Additionally, we will get the log file with the following content:

```
Fri Sep 16 19:32:50 2005
Msg To: UNKNOWN (0x02), Automate ID: 0x00000000
MsgFrom: UNKNOWN (0x0f), Automate ID: 0xcdcdcdcd
Received Msg: (0x0000), Length: 502  Coding type: 0
0f cd 02 ff | 00 00 cd cd | cd cd 00 00 | 00 00 cd cd | cd cd 00 f6 |
...
Start Timer:  (2)
State: 0 -> 1
-------------------------------------------------------
Fri Sep 16 19:32:50 2005
Msg To: UNKNOWN (0x02), Automate ID: 0x00000000
MsgFrom: UNKNOWN (0x0f), Automate ID: 0xcdcdcdcd
Received Msg: (0x0029), Length: 9  Coding type: 0
0f cd 06 ff | 29 00 cd cd | cd cd 00 00 | 00 00 cd cd | cd cd 00 09 | 00 01 00
04 00 | 50 9c 4c 00 | 00
Stop Timer:  (2)
State: 1 -> 2
-------------------------------------------------------
Fri Sep 16 19:32:50 2005
Msg To: UNKNOWN (0x02), Automate ID: 0x00000000
MsgFrom: UNKNOWN (0x0f), Automate ID: 0xcdcdcdcd
Received Msg: (0x002a), Length: 9  Coding type: 0
0f cd 06 ff | 2a 00 cd cd | cd cd 00 00 | 00 00 cd cd | cd cd 00 09 | 00 01 00
04 00 | 50 9c 4c 00 | 00
State: 2 -> 0
-------------------------------------------------------
```

Each record of the log file indicates date and time, message source and destination, message type, message length, message coding type, the content of the message (in hexadecimal code), timer operations, and the state transition information (e.g., "0 -> 1" means a transition from the state S_0 to the state S_1). By looking at this particular log file, we see that the implementation under test behaves as expected. But normally we do not look at the log file if all test cases pass. The real value of the log file is that it is of great help in localizing bugs if a test case fails. Additionally, we could use the log file to check the internal operation of the implementation under test automatically by the tester class. We skipped that step to keep the example simple enough.

5.5.2 Example 2

This example illustrates one of the steps in integration testing of an SIP-based softphone. Imagine that the SIP invite client transaction and the transaction layer dispatcher have undergone complete unit testing. The next normal step would be to integrate them into the final product. Furthermore, imagine that TU and TPL are not yet developed. The only thing we can do is to replace TU and TPL with their imitator classes, named *UA_Test* and *TLI_Test* (TLI stands for Transport Layer Interface), respectively (see the collaboration diagram in Figure 5.17).

The aim of this simple example is to check one particular interaction, illustrated with the collaboration diagram in Figure 5.17. To achieve that goal, we construct the class *UA_Test* that acts as a simple test driver, and the class *TLI_Test* that acts as a simple test stub. Both classes are derived from the class *FiniteStateMachine*. The former class has a single state and a single state transition, whereas the latter has two states and two state transitions.

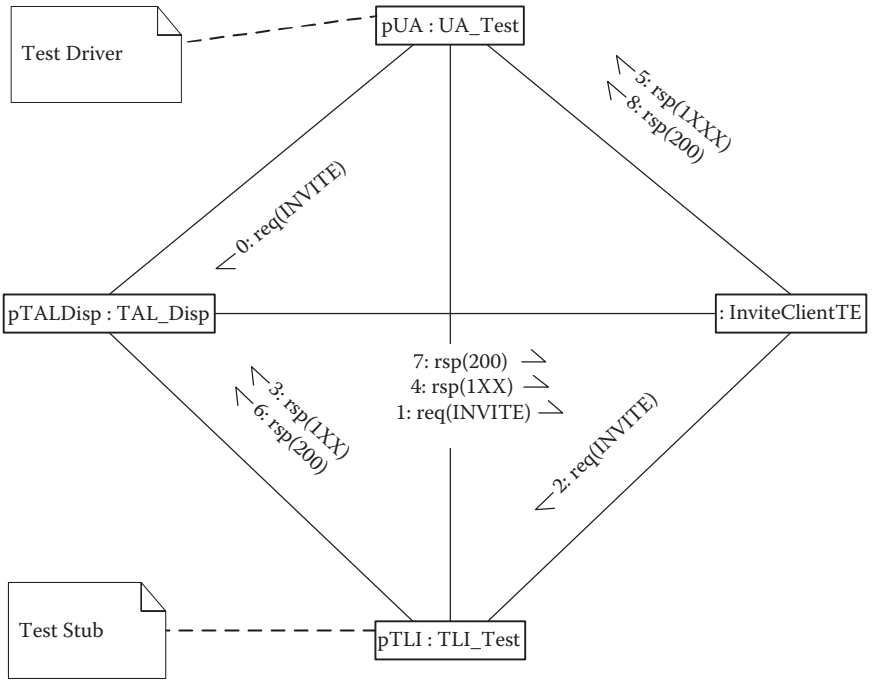


FIGURE 5.17
Example of integration testing collaboration.

The class *UA_Test* declaration file, named *UA_Test.h*, has the following content:

```
#ifndef _UA_Test_FSM_
#define _UA_Test_FSM_
#include "../constants.h"
#include "../kernel/fsm.h"
#include "../message/message.h"

class UA_Test : public FiniteStateMachine {
 int cseq_number;
 Message SIPMsg;
 void SendInviteToTAL();
public:
 enum States { STATE_INITIAL };
 void Evt_Init_TIMER_TINV_EXP();
 void Event_UNEXPECTED();
 // FiniteStateMachine abstract functions
 StandardMessage StandardMsgCoding;
 MessageInterface *GetMessageInterface(uint32 id);
 void SetDefaultHeader(uint8 infoCoding);
 void SetDefaultFSMData();
 void NoFreeInstances();
 void Reset();
 uint8 GetMbxId();
 uint8 GetAutomate();
 uint32 GetObject();
 void ResetData();
public:
 UA_Test();
 ~UA_Test();
 void Initialize();
};
#endif
```

As mentioned above, the class *UA_Test* has a single state, named *STATE_INITIAL*, and a single state transition function, named *Evt_Init_TIMER_TINV_EXP()*. The class *UA_Test* definition file, named *UA_Test.cpp*, has the following content (the parts that are not essential are omitted):

```
#include "UA_Test.h"
#include "../parser/smsgtypes.h"
#include "../parser/smsg.h"
#define SipMessageCoding 0x00
extern char* IPString(unsigned int addr, char* buf, int len);

UA_Test::UA_Test() : FiniteStateMachine(16, 2, 3) {}
UA_Test::~UA_Test() {}

void UA_Test::Initialize() {
 SetState(STATE_INITIAL);
 InitTimerBlock(TIMER_TINV,1,TIMER_TINV_EXPIRED);
 InitEventProc(STATE_INITIAL,TIMER_TINV_EXPIRED,
    (PROC_FUN_PTR)&UA_Test::Evt_Init_TIME_TINV_EXP);
 InitUnexpectedEventProc(STATE_INITIAL,
    (PROC_FUN_PTR)&UA_Test::Event_UNEXPECTED);
 StartTimer(TIMER_TINV);
}

void UA_Test::Evt_Ini_TIMER_TINV_EXP() {
 SendInviteToTAL();
}
```

```
void UA_Test::SendInviteToTAL(){
 char temp[10];
 char szHostName[255];
 hostent* HostData;
 uint8* recmsg;
 sip_t *mes;
 ...
 PrepareNewMessage(0x00,INVITE);
 SetMsgToAutomate(TAL_Disp_FSM);
 SetMsgToGroup(INVALID_08);
 SetMsgObjectNumberTo(0);
 AddParam((SIP_RAW_MESSAGE, SIPMsg.getLastMessage().length(),
    (uint8*) SIPMsg.getLastMessage().c_str());
 AddParamDWord((SIP_PARSED_MESSAGE, (unsigned long) mes);
 SendMessage(TAL_Disp_FSM_MBX);
}
...
```

The function *Initialize()* sets the FSM initial state, initializes the timer *TIMER_TINV* to a 1-s delay, sets the state transition functions, and starts the timer *TIMER_TINV*. When the timer expires, the state transition function *Evt_Init_TIMER_TINV_EXP()* is called. This function sends the *INVITE* message to the transaction layer dispatcher (*TAL_Disp*) by calling the function *SendInviteToTAL()*, which is very similar to the one given in the Example 1 (see Section 5.5.1). Further on, the *INVITE* message is routed toward the test stub class *TLI_Test*.

The class *TLI_Test* declaration file, named *TLI_Test.h*, has the following content (the parts that are not essential are omitted):

```
#ifndef _TLI_Test_FSM_
#define _TLI_Test_FSM_
#include "../constants.h"
#include "../kernel/fsm.h"
#include "../message/message.h"

class TLI_Test : public FiniteStateMachine {
 ...
 Message SIPMsg;
 sip_t *mes;
 // Message management functions
 void Send1XXToTAL();
 void Send2XXToTAL();
public:
 enum States {
  STATE_INITIAL,
  STATE_1XX_SENT
};
 void Evt_Init_INVITE_T();
 void Evt_1XXSent_TIMER_T2XX_EXP();
 void Event_UNEXPECTED();
// FiniteStateMachine abstract functions
 ...
public:
 TLI_Test();
 ~TLI_Test();
 void Initialize();
};
#endif
```

As mentioned above, the class *TLI_Test* has two states, named *STATE_INITIAL* and *STATE_1XX_SENT*, and two state transition functions, named *Evt_Init_INVITE_T()* and *Evt_1XXSent_TIMER_T2XX_EXP()*. The class *TLI_Test* definition file, named *TLI_Test.cpp*, has the following content (the parts that are not essential are omitted):

```
#include "TLI_Test.h"
#define SipMessageCoding 0x00
extern char* IPString(unsigned int addr, char* buf, int len);
TLI_Test::TLI_Test() : FiniteStateMachine(16, 2, 3) {}
TLI_Test::~TLI_Test() {}

void TLI_Test::Initialize() {
 char szHostName[255];
 hostent* HostData;
 SetState(STATE_INITIAL);
 InitTimerBlock(TIMER_T2XX,2,TIMER_T2XX_EXPIRED);
 InitEventProc(STATE_INITIAL,INVITE,
 (PROC_FUN_PTR)&TLI_Test::Evt_Init_INVITE_T);
 InitEventProc(STATE_1XX_SENT,TIMER_T2XX_EXPIRED,

(PROC_FUN_PTR)&TLI_Test::Evt_1XXSent_TIME_T2XX_EXP);
 InitUnexpectedEventProc(STATE_INITIAL,
 (PROC_FUN_PTR)&TLI_Test::Event_UNEXPECTED);
 // Problem specific part
 ...
}

void TLI_Test::Evt_Init_INVITE_T() {
 Send1XXToTAL();
 StartTimer(TIMER_T2XX);
 SetState(STATE_1XX_SENT);
}

void TLI_Test::Evt_1XXSent_TIMER_T2XX_EXP() {
 Send2XXToTAL();
}

void TLI_Test::Send1XXToTAL(){
 uint8* recmsg;
 recmsg = GetParam(SIP_RAW_MESSAGE);
 ...
 SIPMsg.makeResponse("100","Trying",responseBody,0);
 PrepareNewMessage(0x00,RESPONSE_1XX_T);
 SetMsgToAutomate(TAL_Disp_FSM);
 SetMsgToGroup(INVALID_08);
 SetMsgObjectNumberTo(0);
 AddParamDWord((SIP_PARSED_MESSAGE, (unsigned long) mes);
 SendMessage(TAL_Disp_FSM_MBX);
}

void TLI_Test::Send2XXToTAL(){
 SIPMsg.makeResponse("200","OK",responseBody,0);
 PrepareNewMessage(0x00,RESPONSE_2XX_T);
 SetMsgToAutomate(TAL_Disp_FSM);
 SetMsgToGroup(INVALID_08);
 SetMsgObjectNumberTo(0);
 AddParamDWord((SIP_PARSED_MESSAGE, (unsigned long) mes);
   SendMessage(TAL_Disp_FSM_MBX);
}
...
```

The function *Initialize()* sets the initial state, initializes the timer *TIMER_T2XX* to a 2-s delay, sets the state transition functions, and finishes with some problem-specific initializations. The state transition function *Evt_Init_INVITE_T()*, triggered by the reception of the message *INVITE*, sends the preliminary response *100 (Trying)* by calling the function *Send1XXToTAL()*, starts the timer *TIMER_T2XX*, and changes its state to *STATE_1XX_SENT*. The state transition function *Evt_1XXSent_TIMER_T2XX_EXP()*, triggered with the expiration of the timer *TIMER_T2XX*, sends the final response *200 (OK)* by calling the function *Send2XXToTAL()*.

The content of the main module, named *test_main.cpp*, is as follows (the parts that are not essential are omitted):

```
#include <conio.h>
#include "kernel/fsmsystem.h"
#include "kernel/logfile.h"
#include "NewSIP/TAL_Disp.h"
#include "Test/UA_Test.h"
#include "Test/TLI_Test.h"
#include "NewSIP/InvClientTE.h"
 FSMSystemWithTCP *pSys;
 LogFile *lf;
 TAL_Disp* pTALDisp;
 TLI_Test* pTLI;
 UA_Test* pUA;
 InviteClientTE* pInviteCltTE[NUMBER_OF_TES];
 DWORD thread_id;
 HANDLE thread_handle;
 ...
DWORD WINAPI SystemThread(void *data){
 FSMSystem *sysAutomate = (FSMSystem *)data;
 sysAutomate->Start();
 return 0;
}
int init(){
 pSys = new FSMSystemWithTCP(11,11);
 pTALDisp = new TAL_Disp();
 pTLI = new TLI_Test();
 pUA = new UA_Test();
 for (int i = 0; i < NUMBER_OF_TES; i++){
 pInviteCltTE[i]= new InviteClientTE();
 }
 uint8 buffClassNo = 4;
 uint32 buffsCount[4] = { 50, 50, 50, 50 };
 uint32 buffsLength[4] = { 1025, 1025, 1025, 1025};
 pSys->InitKernel(buffClassNo, buffsCount, buffsLength, 1);
    lf = new LogFile("log.log", "log.ini");
 LogAutomateNew::SetLogInterface(lf);
 pSys->Add(pTALDisp, TAL_Disp_FSM, 1, false);
 pSys->Add(pInviteCltTE[0], InviteClientTE_FSM, 10, true);
 pSys->Add(pTLI, TLI_Test_FSM, 1, false);
 pSys->Add(pUA, UA_Test_FSM, 1, false);
 for (i = 1; i < NUMBER_OF_TES; i++){
    pSys->Add(pInviteCltTE[i], InviteClientTE_FSM);
 }
 thread_handle = CreateThread(NULL, 0, SystemThread, pSys,
 THREAD_PRIORITY_ABOVE_NORMAL, &thread_id);
 return 1;
}
...
```

```
void main (void){
 parser_init();
 init();
while(!kbhit());
 exit_app();
}
```

As a result of the execution of the main module, we get the log file with nine records that correspond to the messages that are exchanged between implementations under test (the transaction layer dispatcher and SIP invite client transaction), test driver (*UA_Test*), and test stub (*TLI_Test*). This file is very similar to the one given in Example 1 (see Section 5.5.1) but three times longer, and hence is not included here.

Test automation of integration tests based on log files is possible for simple collaborations like the one shown in this example, although it may be cumbersome. However, if we must deal with more complex collaborations that evolve concurrently, this approach is hardly applicable. Using log files in such situations would normally require human intervention for checking the results of the integration tests. Generally, we should try to use the style of unit testing based on the automatic checking of results (see Section 5.1), even for the integration of the parts of the system.

5.6 Further Reading

The reader can find more information related to this chapter in the references. The research by Berard et al. (2001) contains comprehensive coverage of the state-of-the-art model-checking techniques and tools. Newborn (2001) provides detailed information on the theorem prover THEO used in Section 5.3. Hoare (1985) wrote the famous book on CSP (nowadays, it is also available online for free as a PDF). The study of Sun (2009) is about PAT. Fischer et al. (2006) and Canepa et al. (2008) are the original papers on leader election algorithms, which appeared in the examples in Sections 5.3.2.3.3, 5.3.2.3.4, and 5.3.2.3.5. Popovic et al. (2001) provide a software maintenance case study in the area of communication protocol engineering. The research by Popovic and Velikic (2005) contains more information on the generic test case generator used in Section 5.5. Woit (1994; Chapter 3 and Section 3.1.1, in particular) provides more information on the reliability estimation model used in Section 5.5.

References

Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, Ph., and McKenzie, P., *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer-Verlag, Berlin, 2001.

Canepa, D. and Gradinariu Potop-Butucaru, M., Stabilizing token schemes for population protocols, arXiv:0806.3471v1 [cs.DC], 2008. Available online at https://arxiv.org/abs/0806.3471v1 (accessed June 28, 2017).

Fischer, M.J. and Jiang, H., "Self-stabilizing leader election in networks of finite-state anonymous agents," *Proceedings of the 10th Conference on Principles of Distributed Systems*, Vol. 4305 of LNCS, Springer, pp. 395–409, 2006.

Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, 1985. Available online at http://usingcsp.com/cspbook.pdf (accessed June 28, 2017).

Newborn, M., *Automated Theorem Proving*, Springer-Verlag, New York, 2001.

Popovic, M., Atlagic., B., and Kovacevic, V., "Case study: A maintenance practice used with real-time telecommunication software," *Journal of Software Maintenance and Evolution: Research and Practice*, John Wiley & Sons, West Sussex, No. 13, pp. 97, 2001.

Popovic, M. and Velikic, I., "A generic model-based test case generator," *Proc. IEEE International Conference and Workshop on Engineering of Computer Based Systems*, Greenbelt, MD, April 4–7, 2005.

Sun, J., Liu, Y., Dong, J.S., Pang, J., "PAT: Towards Flexible Verification under Fairness," *Proceedings of the 21th International Conference on Computer Aided Verification (CAV'09)*, Vol. 5643 of LNCS, Springer, pp. 709–714, 2009.

Woit, D.M., "Operational profile specification, test case generation, and reliability estimation for modules," Ph.D. thesis, Queens University Kingstone, Ontario, Canada, February 1994.

6

FSM Library

The purpose of this chapter is to familiarize the reader with an example of a real-world library for making families of communication protocols. Although it is not perfect, it is in use and evolving. The main argument against it may be that there are too few C++ classes with too many function members. Alternately, this disadvantage is a tradeoff for a rather simple API, which is quite easy to learn and use.

6.1 Introduction

The FSM Library described in this book was created to be used as a working environment for the implementation of groups of communication protocols. The programmer has two basic classes at his or her disposal, namely, *FSMSystem* and *FiniteStateMachine*. The class *FSMSystem* models a platform for a group of communication processes (otherwise called finite state machines or automata). An instance of this class interconnects individual communication processes by handling all of the resources needed for the operation of individual finite state machines.

The class *FiniteStateMachine* models a generic communication process (i.e., communication protocol). Each individual communication protocol is represented by an instance of this class. The implementation of a particular communication protocol is narrowed down to writing state transition functions in C++. The transition function comprises procedures that process the message received in a given FSM state. This processing results in a transition to a new FSM state and the optional generation of corresponding outgoing messages. All state transition functions must be defined for all the finite state machines registered to a single FSM system (an instance of the class *FSMSystem*). Additionally, all the FSM system run-time elements must be initialized properly before they can be successfully started.

The relationship between the classes *FSMSystem* and *FiniteStateMachine* is symbiosis—one cannot operate without the other. The FSM system clearly represents an infrastructure, or an unused platform. In reality, an FSM system is always used so that at least a couple of finite state machines are registered to it, together representing a group of finite state machines. Because of this, and in order to achieve simplicity and brevity, we frequently use

the term "FSM system" as a synonym for the group of automata, assuming that some individual automata are actually registered to it, and vice versa. Although an instance of the class *FiniteStateMachine* cannot operate on its own, we simply refer to it as a "finite state machine."

6.2 Basic FSM System Components

The FSMSystem Library is written in C++ using an object-oriented approach. The basic components are written as C++ classes that provide functionality of both individual finite state machines and a group of finite state machines. These classes are the following:

- *FiniteStateMachine*
- *FSMSystem*

A class can inherit the functionality of a single finite state machine by specializing the base class *FiniteStateMachine*. The programmer implements this class by writing the real functions for those declared as virtual, by adding new problem-specific functions (e.g., state transition functions), and by optionally overriding the inherited functions to redefine the functionality of the base class.

A class can inherit the functionality of a group of finite state machines by specializing the class *FSMSystem*. Normally, this class is simply instantiated as an oracle of a group of finite state machines.

6.2.1 Class *FSMSystem*

An instance of the class *FSMSystem* is an object representing a finite state machine system, i.e., a group of finite state machines (a group of automata). The protected attributes of this class represent the resources available for all the automata included in a group of automata. The basic task of this class is the initialization and management of FSMs, buffers (memory zones), messages, and timers. During a normal lifecycle of an instance of the class *FSMSystem*, its user typically performs the following steps or operations:

- Create FSM system
- Initialize FSM system
- Start FSM system
- Stop FSM system

In the list above, the idiom "FSM system" represents an instance of the class *FSMSystem*.

6.2.1.1 FSM System Initialization

The initialization of the FSM system consists of the following steps:

- Create the FSM system—see the constructor *FSMSystem()*.
- Create and initialize individual finite state machines—see the constructor *FiniteStateMachine()*.
- Add individual finite state machines to the FSM system.
- Initialize the FSM system.
- Start FSM system logging.

The constructor *FSMSystem()* requires two parameters:

- The number of types of finite state machines
- The number of mailboxes

Individual instances of the class *FiniteStateMachine* can be added to the FSM system by using one of two the possible functions:

```
void Add(ptrFiniteStateMachine object, // Automata instance address
 uint8 automataType, // Automata type
 uint32 numOfObjects, // Number of instances
 bool useFreeList = false); // List of free automata

void Add(ptrFiniteStateMachine object, // Automata instance address
 uint8 automataType); // Automata type
```

The first of the overloaded functions above is used to add the first finite state machine of each type. The other instances of the same type are added using the second function.

The initialization of the FSM system kernel is performed by calling the following function:

```
void InitKernel(uint8 buffClassNo, // Number of different types
 uint32 *buffersCount, // Number of buffers per type
 uint32 *buffersLength, // Buffer lengths per type
 uint8 numOfMbxs=0, // Number of mailboxes
 TimerResolutionEnum timerRes = Timer1s); // Timer  resolution in ms
```

The parameters of the function *InitKernel* specify the number of buffer types, the numbers of the instances of different types, their sizes, the number of mailboxes to be used by the automata in a group, and the basic timer resolution. The default number of mailboxes is 0. The default basic timer resolution is 1 sec (just as an example, it can be much smaller, e.g., 10 ms).

The FSM system logging functionality provides message content recording in a sequence resulting from the evolution of the FSM system. These messages are recorded automatically into a file created at the FSM system startup. The file *log.ini* is optional and is used to define textual titles (names) of the

messages exchanged among the finite state machines included in the corresponding FSM system. If *log.ini* file is defined, the message binary codes are substituted by the corresponding message names, thus making the log files human readable. On Windows® machines, the *log.ini* file must be placed in the system folder (*c:\winnt* or *c:\windows*). The format of this file is as follows:

```
[AUTOMATA]
1=AUTOMATA1_FSM
2=AUTOMATA2_FSM
SequenceNumber=AUTOMATA_TYPE
[MESSAGES]
0=0xe000,MSG_1,0
1=0xe002,MSG_2,0
SequenceNumber=MSG_CODE,TEXT_TITLE,0
```

A typical example is as follows:

```
#define NO_BUFFERS 3
#define NO_AUTOMATA_1 5
#define NO_AUTOMATA_2 9
...

// Definition of buffers: three types, where number of buffers per type
// is 50, 30, and 20, and their lengths are 128, 256, and 512 bytes,
// respectively.
uint8 buffClassNo = NO_BUFFERS;
uint32 buffersCount[NO_BUFFERS] = {50,30,20};
uint32 buffersLength[NO_BUFFERS] = {128,256,512};

// Create FSM system that has two automata types and uses
// two mailboxes (one mailbox per each automata type)
FSMSystem *fsmSystem = new FSMSystem(2,2);

// Create individual automata
Automata1 *automata1 = new Automata1[NO_AUTOMATA_1];
Automata2 *automata2 = new Automata2[NO_AUTOMATA_2];

// Add individual automata to FSM system and implicitly initialize each
// automata instance by calling its function Initialize(). This call is
// made from the function Add.
fsmSystem->Add(&automata1[0],AUTOMATA1_FSM,NO_AUTOMATA_1,false);
for((i=1; i<NO_AUTOMATA_1; i++))
 fsmSystem->Add(&automata1[i],AUTOMATA1_FSM);

fsmSystem->Add(&automata2[0],AUTOMATA2_FSM,NO_AUTOMATA_2,true);
for((i=1; i<NO_AUTOMATA_2; i++))
 fsmSystem->Add((&automata2[i],AUTOMATA2_FSM);

// Initialize kernel
fsmSystem->InitKernel(buffClassNo,buffersCount,buffersLength,2);
// Create and set logging system (log file name, message definition file)
lf = new LogFile("log.log", "log.ini");
LogAutomataNew::SetLogInterface(lf);
...
```

The example above starts with the definition of the number of buffer types. In this example, three buffer types are defined (i.e., small, medium, and large buffers) by setting the symbolic constant *NO_BUFFERS* value to 3. Next, we

define the number of instances of two automata types by setting the values of symbolic constants *NO_AUTOMATA_1* to 5 and *NO_AUTOMATA_2* to 9. This means that five instances of the first automata type and nine instances of the second automata type will exist in the group of automata we are going to create.

Next, the program paragraph defines the number of buffers, as well as their size, for each buffer type. Fifty small buffers of size 128 bytes, thirty medium buffers of size 256 bytes, and twenty large buffers of size 512 bytes would be used. The number of buffer types is stored in the variable *buffClassNo*. The number of buffers of each type and their lengths are stored in the arrays *buffersCount* and *buffersLength*.

We then create the FSM system by calling the constructor of the class *FSMSystem*. This constructor has two parameters: the number of automata types and the number of mailboxes to be used by the system for its own purposes. Next, we create two groups of automata of two different types. In this program, these groups are represented as arrays of instances of classes, namely, the classes *Automata1* and *Automata2*. In this example, we assume that these classes have already been defined by extending the base class *FiniteStateMachine*.

After creating two groups of automata of different types, all the automata are added to the already created FSM system. The first instance of each automata type is added by calling the overloaded function *Add* with the first type of signature, which specifies the instance address, the instance type, the total number of instances of this type, and the indicator specifying if a list of free automata of this type exists or not. The rest of the instances are added by calling the overloaded function *Add* with the second type of signature, specifying just the instance address and its type.

The first automata type in this example does not have a list of free automata, whereas the second type does have a list of free automata. This means that the instance of the second automata type can be viewed as a pool of resources of the same type. They may be dynamically allocated to be engaged in a certain communication scenario. When a programmer decides to use this opportunity, they must provide the function *NoFreeInstance*, which is called when the dynamic allocation request cannot be satisfied, because no more free automata instances of that type are found.

The FSM system is initialized by simply calling its function *InitKernel*. The parameters of this function specify the number of buffer types, the number of buffers of each type, their sizes, and the number of mailboxes to be used for FSMs. Normally, we use one mailbox per automata type. This is not a restriction imposed by the class *FSMSystem*, it is simply a convention. Other arrangements are also allowed; for example, we can create more mailboxes for messages of different priorities, or we can create additional mailboxes dedicated to communication between the given groups of automata types. Most generally, we can use mailboxes as queues of any kinds of messages. Because the last parameter of the function *InitKernel* is omitted, the timer resolution is set to its default value (1 sec, in this example).

At the end of this example, we create and set the logging system by calling its constructor *LogFile* and the function *SetLogInterface*, respectively. The parameters of the constructor specify the name of the log file (*log.log*) as well as the name of the file containing the textual names of the messages (*log.ini*). The parameter of the function *SetLogInterface* specifies the logging system interface, which generally is a file. In this example, the disk file is named *log.log* but it could be any file, including special files representing devices handled by the corresponding device drivers, such as */dev/lpt* or */dev/com1*.

6.2.1.2 FSM System Startup

The FSM system is started by calling its function *Start*. Most frequently, this function is called by the thread assigned to the FSM system. Here is an example:

```
DWORD WINAPI FsmSystemThreadFunc((void* param)){
 try {
  fsmSystem->Start();
 }
 catch(...){
  OutputDebugString('Exception – terminating FSM system\n');
  return 0;
 }
 OutputDebugString('FSM system terminated\n');
 return 0;
}
...

// Somewhere in the main function
DWORD fsmSystemThreadId;
CreateThread(NULL,0,FsmSystemThreadFunc,0,0,fsmSystemThreadId);
...
```

In the example above, we start the FSM system by calling its function *Start* from the thread function *FsmSystemThreadFunction*. We assume that thread has already been created and that its identification is stored in the variable *fsmSystemThreadId*.

6.2.2 Class *FiniteStateMachine*

All the automata added to the FSM system are implemented by extending the base class *FiniteStateMachine*. This class defines a set of virtual functions that must be defined by the programmer. These functions are as follows:

```
MessageInterface *GetMessageInterface(uint32 id);
void SetDefaultHeader(uint8 infoCoding);
uint8 GetMbxId();
uint8 GetAutomata();
void SetDefaultFSMData();
void NoFreeInstances();
void Initialize();
```

The following example illustrates the most frequently used definitions of *FiniteStateMachine* functions. A detailed description of all the functions is given in Section 6.8.

```
// This function returns the message interface for the given interface ID.
// It is assumed that standardMsgCoding is defined as:
// StandardMessage standardMsgCoding;
MessageInterface *Automata::GetMessageInterface(uint32 id){
 switch(id){
  case 0x00:
  return &standardMsgCoding;

  // Other definitions
  // case 0x01:
  // case 0x02:
 }
 throw TErrorObject(__LINE__,__FILE__,0x01010400);
}

// This function fills in the message header.
void Automata::SetDefaultHeader(uint8 infoCoding){
 SetMsgInfoCoding(infoCoding);
 SetMessageFromData();

}
// This function defines the mailbox number (ID) to be used as default
// by the automata of the type defined by this class.
uint8 Automata::GetMbxId(){
 return AUTOMATA_MB_ID;
}

// This function returns the number (ID) which identifies the automata
// type defined by this class.
uint8 Automata::GetAutomate(){
 return AUTOMATA_TYPE_ID;
}

// This function sets the values of the instance attributes.
void Automata::SetDefaultFSMData(){
 attribut1 = VALUE_1;
 attribut2 = VALUE_2;
}

// This function is called if there are no more free automata of this
// type. It may be used if the instances of this class have been added to
// the FSM system with the parameter useFreeList set to value true.
void Automata::NoFreeInstances(){
 // The activity if there are no free automata of this  type.
}

 // This function defines state transition functions and timers to be used
 // by the automata of this type. It is called by the function Add, which
 // is used to add an automata instance to the given FSM system.
 // It is assumed that state transition functions are declared and defined
 // elsewhere.
void Automata::Initialize(){
 // Here we place a series of initializations:
 // InitEventProc(uint8 state, uint16 event,  PROC_FUN_PTR fun);
 // InitUnexpectedEventProc(uint8 state, PROC_FUN_PTR  fun);
 // InitTimerBlock(uint16 timerId, uint32 timerCount,  uint16 signalId);
```

```
 InitEventProc(IDLE, MSG_SEND, (PROC_FUN_PTR)  &Automata::Idle_MsgSend);
 InitEventProc(IDLE, MSG_RCV,  (PROC_FUN_PTR)  &Automata::Idle_MsgReceive);

 InitEventProc(SEND, MSG_NEW,  (PROC_FUN_PTR)  &Automata::Send_MsgNew);
 InitEventProc(SEND, MSG_END,  (PROC_FUN_PTR)  &Automata::Send_MsgEnd);
 InitEventProc(IDLE, T200_CODE,(PROC_FUN_PTR)  &Automata::T200Expired);

 InitUnexpectedEventProc(IDLE, (PROC_FUN_PTR)  &Automata::Idle_Unexpected);
 InitUnexpectedEventProc(SEND, (PROC_FUN_PTR)  &Automata::Send_Unexpected);

 InitTimerBlock(T200,T200_VALUE,T200_CODE);
}
```

In the example above, we would like to create the class *Automata* that models one type of finite state machines (automata). The definition of the class comprises the definitions of its function members. The function member *GetMessageInterface* returns the object that embodies the coding of messages to be used by the instances of the class *Automata*. In this example, it is an instance of the class *StandardMessage*.

The member function *SetDefaultHeader* is used to automatically fill in the message header defaults. Normally, these are the data about the automata instance that has created the message to send to some other automata instance. In this example, it uses the function *SetMsgInfoCoding* to specify the type of coding to be applied. It also uses the function *SetMessageFromData* to specify the type of originating automata instance, the identification of the group to which the automata instance belongs, and the identification of the originating automata instance.

The member function *GetMbxId* returns the identification of the mailbox used by the automata instance of this type. In this example, it is the value of the symbolic constant *AUTOMATA_MBX_ID*. The member function *GetAutomata* returns the identification of the automata type. It is the value of the symbolic constant *AUTOMATA_TYPE_ID*. The member function *SetDefaultFSMData* is used by the automata instance to set its specific data before it commences its normal operation. In this example, *attribute1* is set to the value *VALUE_1* and *attribute2* is set to the value *VALUE_2*.

The member function *NoFreeInstances* can be used to specify the action to be performed, if no more free automata instances of this type are found, e.g., to make a small system restart, allocate some additional automata instances, and so on. This mechanism is available to the programmer if the instances of automata have been added (function *Add*) to the FSM system with the parameter *useFreeList*, set to the value *true*.

The member function *Initialize* is used to define automata state transition functions and timers (referred to as timer blocks throughout the FSM Library documentation) to be used by the automata. The FSM Library distinguishes two types of events, expected and unexpected, and allows the programmer to specify the corresponding event handlers, which are just specialized C++ functions. These handlers are defined by calling the registration functions, namely, the function *InitEventProc* for expected events and the function *InitUnexpectedEventProc* for unexpected events. The parameters of both

TABLE 6.1

Example of a State Transition Table

	MSG_RCV	MSG_ SEND	T200_CODE	MSG_ NEW	MSG_ END	?
Idle	Idle_ MsgReceive	Idle_ MsgSend	T200Expired			Idle_ Unexpected
Send				Send_ MsgNew	Send_ MsgEnd	Send_ Unexpected

of these functions specify the state code, the event (message) code, and the pointer to the event handler.

In this example, we have defined seven automata state transition functions altogether, five of them triggered by expected events and two triggered by unexpected events. The part of the automata shown in the example has two states, *IDLE* and *SEND*. The expected events in the state *IDLE* are *MSG_SEND, MSG_RCV*, and *T200_CODE*. The corresponding event handlers are *Idle_MsgSend*, *Idle_MsgReceive*, and *T200Expired*, respectively. Two legible events exist in the state *SEND, MSG_NEW* and *MSG_END*. The corresponding handlers are *Send_MsgNew* and *Send_MsgEnd*. The unexpected event handler for the state *IDLE* is *Idle_Unexpected* whereas for the state *SEND* it is *Send_Unexpected*. The corresponding state transition table is shown in Table 6.1.

The timers are initialized by calling the function *InitTimerBlock*. The parameters of this function specify the unique timer identification, its duration (as the number of basic timer resolution units), and the code of the message sent when the timer expires. In the example above, these are the symbolic constants *T200*, *T200_VALUE*, and *T200_CODE*.

To sum, automata states and attributes are defined in accordance with the problem at hand. The state transition function, referred to as the event handler, is called upon the reception of a given message in a given state, as defined by the function *Initialize*. Each event handler is defined as a class member function responsible for handling a given event.

The timers to be used by the automata are also defined by the function *Initialize*. This is done by calling the function *InitTimerBlock*, which, in turn, creates the internal kernel timer block (essentially a program object) and fills in its identification, duration, and corresponding timer message code.

6.3 Time Management

In Section 6.2, automata timers are initialized during the FSM system startup by the function *Initialize*. The automata type that uses timers in its regular operation manages them through the corresponding FSM Library API

functions, which maintain the internal kernel object behind the scenes. The API functions are the following:

```
void InitTimerBlock(uint16 tmrId,uint32 count,uint16 signalId);
void StartTimer(uint16 tmrId);
void StopTimer(uint16 tmrId);
void RestartTimer(uint16 tmrId)
bool IsTimerRunning(uint16 tmrId);
```

The function *InitTimerBlock* is used to define (initialize) the timer. Its parameters specify the unique timer identification, its duration as a multiple of the basic timer resolution unit, and the code of the message sent to the automata mailbox when the timer expires. This is explained in the previous section. Notice that each timer has the unique identification *tmrId* used as a parameter of all the API functions to identify the timer.

Each API function represents a primitive timer operation. The function *StartTimer* is used to start the timer, the function *StopTimer* stops the timer, the function *RestartTimer* restarts the timer, and the function *IsTimerRunning* is used to check if the timer is running or not.

The following example illustrates the usage of these primitives:

```
if(!IsTimerRunning(T200)){
 StartTimer(T200);
}
else
 StopTimer(T200);
...
```

A normal timer life cycle has the following phases:

- Define, i.e., initialize, the timer.
- Use the timer by alternative application of the following primitives:
  - *Start* (applicable if the timer is not running, meaning it was either newly defined or previously stopped)
  - *Stop* (applicable if the timer is running)
  - *Restart* (logically equivalent to *Stop* plus *Start*)
  - *IsTimerRunning* (returns true if it does; otherwise, it returns false)

6.4 Memory Management

Because the main application of the FSM Library is in real-time systems, efficient memory allocation must be provided. The FSM Library does not rely on a hosting operating system because some of the operating systems suffer from a memory fragmentation problem. Furthermore, in some applications

on bare machines, the operating system may not even be available. Because of that, memory management is one of the main functions of the FSM Library.

The working memory is partitioned into certain zones referred to as **buffers**. The programmer defines the number of different buffer types, the number of buffers of each type, and their sizes. The programmer specifies this data as parameters of the function *InitKernel* (see Section 6.8.4) and the FSM Library kernel, in turn, creates them as its own internal objects.

The buffers are most frequently used indirectly through message management (message create, send, receive, and similar operations) and timer operations (timer definition and usage operations). Besides this indirect buffer usage, the buffers can be managed directly, if needed, through the following API functions:

```
uint8 *GetBuffer(uint32 length);
void  RetBuffer(uint8 *buff);
bool  IsBufferSmall(uint8 *buff,uint32 length);
uint32 GetBufferLength(uint8 *buff);
```

The programmer requests a buffer by calling the function *GetBuffer*. The parameter of this function is the minimal size of the desired buffer. All the buffers provided by the kernel must be returned by calling the function *RetBuffer*. Untidy memory management can cause buffer loss, commonly referred to as **memory leak**, which may cause irregular kernel operation and a system crash.

Besides memory allocation (*malloc*) and *free* primitives, two additional primitives provide the information about the buffer already allocated to the finite state machine. The function *IsBufferSmall* checks if the buffer size is smaller than the value of its parameter. If yes, it returns *true*, otherwise, it returns *false*. Another function, named *GetBufferLength*, returns the buffer size in octets (bytes).

The following example illustrates the usage of the buffer management primitives:

```
// We define two buffer types, small and large.
// There are ten small buffers and fifteen large buffers.
// The small buffer size is 128 bytes. The large buffer size is
// 256 bytes.
uint8 buffClassNo = 2;
uint32 buffersCount[2] = {10,15};
uint32 buffersLength[2] = {128,256};
...

// Kernel initialization (noMBX is irrelevant in this example)
fsmSystem->InitKernel(buffClassNo,buffersCount,buffersLength,noMBX);
...

uint32 bufferLength;
uint8 *pointer = GetBuffer(100);
if((IsBufferSmall(pointer,129)){
 RetBuffer(pointer);
 pointer = GetBuffer(129);
}
if((pointer != NULL))
 bufferLength = GetBufferLength(pointer);
...
```

In the example above, we first define two buffer types—small and large—by calling the function *InitKernel*. Its fourth parameter (*noMBX*, the number of the mailboxes) is not relevant for this example. The rest of the program illustrates the usage of the FSM Library's buffer management functions. First, the program asks for a buffer not smaller than 100 bytes, then it checks if this buffer is smaller than 129 bytes. If yes, it returns the allocated buffer and requests a new one not smaller than 129 bytes (in this example, it will get one large buffer of size 256 bytes). At the end, the program checks if the pointer is defined, which also means that it points to a certain buffer. If it is defined, the program asks for its size by calling the function *GetBufferLength*.

6.5 Message Management

The main communication among individual automata included in the FSM system is achieved through the messages exchanged through the mailboxes typically assigned to individual automata. The message sent from the originating automata instance towards the destination automata instance is placed temporarily in the mailbox assigned to the destination automata instance. There, it waits to be taken over and subsequently processed by the destination automata instance (process).

As already mentioned, a **mailbox** is a message queue that can contain messages for any automata type, thus it does not need to be assigned to some particular automata type. In contrast to a typical paradigm, it can be used as a general message queue shared by more destination automata. Essentially, in such a paradigm, the source automata instance can put the message in any mailbox hosted by the FSM system, and it will eventually be delivered to its proper destination.

This message routing and delivery is performed automatically by the FSM system and is hidden from the automata, which are just service users. The FSM system has an abstraction of the mailbox from which it takes messages, one at a time (mailbox abstraction provides buffering functionality by employing the FIFO memory type). Upon the reception of each individual message, the FSM system consults the message header to determine the destination automata instance and passes the message to it. The destination automata instance looks up the message code and, based on the current automata state, calls the appropriate automata state transition function.

Message reception is completely transparent for the programmer writing the program code for the finite state machine. The above mechanism is absolutely hidden from them. The programmer must simply accept that the message reception and its classification are done automatically by the system. They just write the message processing functions that are called automatically by the system upon the reception of the corresponding message.

The API functions can be partitioned into two groups:

- The functions that work with the received message.
- The functions that work with the new message that must be prepared and sent.

The functions in the first group are used to provide the information about the originating automata instance. The source of this information is the message header and the values of the message parameters. The functions in the second group provide primitives needed to make and send a message:

- Buffer allocation (indirect call to *GetBuffer* primitive)
- Filling the message header with the data about the originating automata instance
- Adding the message parameters and setting them to the given values
- Sending the message to the mailbox assigned to the destination automata instance

The messages may be sent only from a finite state machine or a FSM system. Note that during normal system operation, a FSM system does not send any messages. In this context, a finite state machine is an instance of the class *FiniteStateMachine*, or a class derived from it, and an FSM system is an instance of the class *FSMSystem*.

Example 1:

```
// Get parameter of type PARAM_1 from the received message.
// The size of PARAM_1 is WORD.
WORD word;
GetParamWord(PARAM_1,word);

// Get parameter of arbitrary size. Maximum size for StandardMessage is
// 256 bytes. If that is not sufficient, a programmer must derive a new
// class and redefine its functions.
uint8    *pointer;
uint8    text[300];
uint8    msgLength;

pointer = GetParam(TEXT);
if(pointer != NULL){
 // StandardMessage format: bytes 1 and 2 contain parameter name,
 // byte 3 contains parameter length in bytes,
 // byte 4 and further contain the parameter itself.
 memcpy(text,pointer+3,*((pointer+2)));

 // Make a string by placing null at the end of  character array.
 memset(text+(*((pointer+2))),0x00,1);
}
```

The example above shows how the programmer can get a parameter from the *current* message. A current message is the last message received by the

automata instance, i.e., it is the last message taken from the mailbox and assigned to the automata instance for processing. The parameter size is *WORD* (2 bytes). First, the programmer declares the variable *word* in which he wants to store the parameter value.

The message can contain many parameters, and therefore the programmer must specify the unique identifier of the parameter they want to get. In this example, the identifier is the value of the symbolic constant *PARAM_1*. Finally, a copy of the desired parameter is provided by calling the API function *GetParam*. The first parameter of this function is the parameter identifier (*PARAM_1*) and the second is the variable (*word*) in which the desired parameter is to be copied.

The second part of the example above demonstrates how the programmer may handle textual parameters of arbitrary size. The *StandardMessage* format prescribes that the first 2 bytes of such a parameter are reserved for the parameter name, the next byte is used for the parameter length (in bytes), and the rest of the bytes in the parameter represent its value. The example shows how a copy of such a parameter can be provided and how a null terminated string can be constructed by adding the NULL character at its end.

Example 2:

```
...
// PrepareNewMessage parameters: buffer size and message type.
PrepareNewMessage(0xAA,MSG_NAME);

// Fill in the message header:
// destination automata type, its ID, and optionally its group ID.
SetMsgToAutomata(AUTOMATA_TYPE);
SetMsgObjectNumberTo(automataId);
SetMsgToGroup(INVALID_08);

// Add parameters: see also other AddParam functions.
AddParamByte(PARAM_1,byte);
AddParamWord(PARAM_2,word);
AddParam(PARAM_3,parameterLength,parameterPointer);

// Send message to the specified mailbox.
SendMessage(AUTOMATA_MBX_ID);
```

The example above shows a common way to construct and send a message. The first step is to call the function *PrepareNewMessage*. The parameters of this function specify the expected buffer size (*0xAA* in this example) and the message name, which also specifies the message type (*MSG_NAME*).

Next, we fill in the message header by calling the following functions:

- *SetMsgToAutomata*: set the destination automata instance type (*AUTOMATA_TYPE*)
- *SetMsgObjectNumberTo*: set the destination automata instance identification (*automataId*)
- *SetMsgToGroup*: set the automata instance group identification (*INVALID_08*)

We then add three message parameters by calling members of the *AddParam* family of functions. The first function shown in the example is *AddParamByte*. Its parameters specify the unique parameter identifier (*PARAM_1*) and the variable containing the value of the parameter to be copied to the corresponding field of the message (*byte*). The second function is *AddParamWord*. Similarly, its parameters specify the parameter identification (*PARAM_2*) and the variable holding its value (*word*). The last function is *AddParam*. The parameters of this function specify the parameter identification (*PARAM_3*), its length (*parameterLength*), and a pointer to it (*parameterPointer*).

At the end of the example above, we send the message by calling the function *SendMessage*. The parameter of this function specifies the destination mailbox identification (*AUTOMATA_MBX_ID*).

Example 3:

```
// Send a message from the FSM system.
uint8 *msg = GetBuffer(messageInfoLength+MSG_HEADER_LENGTH);

// infoBuffer must be properly formatted.
memcpy(msg+MSG_HEADER_LENGTH,infoBuffer,infoBufferLength);

SetMsgFromAutomata(AUTOMATA_TYPE_FROM_ID,msg);
SetMsgFromGroup(INVALID_08,msg);
SetMsgObjectNumberFrom(automataFromId,msg);

SetMsgToAutomata(AUTOMATA_TYPE_TO_ID,msg);
SetMsgToGroup(INVALID_08,msg);
SetMsgObjectNumberTo(automataToId,msg);

SetMsgInfoCoding(0,msg); // 0 = StandardMessage
SetMsgCode(MSG_FROM_SYSTEM_AUTOMATA,msg);
SetMsgInfoLength(infoBufferLength,msg);
SendMessage(AUTOMATA_TO_MBX_ID,msg);
...
```

The example above shows how a message can be created and sent within the FSM system. This process is done through the following steps:

- Allocate a buffer by calling the function *GetBuffer*.
- Copy the information payload.
- Fill in the data about the originating automata instance by calling the function *SetMsgFromAutomata* fill in the originating automata instance type identification (AUTOMATA_TYPE_FROM_ID); by calling the function *SetMsgFromGroup*, fill in the originating automata instance group identification (*INVALID_08*); and by calling the function *SetMsgObjectNumberFrom*, fill in the automata instance identification (*automataFromId*).
- Fill in the data about the destination automata instance. The function *SetMsgToAutomata* sets the destination automata instance type identification (*AUTOMATA_TYPE_TO_ID*), the function

SetMsgToGroup sets the destination automata instance group identification (*INVALID_08*), and the function *SetMsgObjectNumberTo* sets the destination automata instance identification (*automataToId*).

- Finalize the message. The function *SetMsgInfoCoding* sets the type of coding (*StandardMessage*), the function *SetMsgCode* sets the message code (*MSG_FROM_SYSTEM_AUTOMATA*), and the function *SetMsgInfoLength* sets the payload length (*infoBufferLength*).
- Send message by calling the function *SendMessage* with the second type of the signature. The parameters of this function specify the destination mailbox identification (*AUTOMATA_TO_MBX_ID*) and the pointer to the message to be sent (*msg*).

6.6 TCP/IP Support

One of the primary design goals of creating the FSM Library was to support the design of scalable applications based on distributed processing. The FSM Library enables both single-processor and multiprocessor applications. In the former case, all groups of automata execute in a single processor. They share processor resources, such as its processing unit, operating memory, flash, and so on. The automata communicate over the mailboxes placed in the common operating memory.

In the latter case, various groups of automata are deployed on more processors, which can be logically viewed as a multiprocessor system. The groups of automata execute on different processors in parallel and use the mailboxes physically located in separate operating memories. The FSM Library transparently uses the network infrastructure to pass messages among the communicating automata. Most frequently, the communication infrastructure is the TCP/IP technology.

In both cases, the communicating automata are unaware of the real physical infrastructure because the physical details are hidden from them. This is accomplished by providing a unique API. An individual automata instance manages just its timers, buffers, and messages (new and current, i.e., last received). The rest is handled by the FSM Library kernel behind the scenes. This means that the FSM Library inherently provides implicit support for TCP/IP. For example, if an automata instance wishes to send a message to some other automata instance physically located on a different machine, it just prepares the message and calls the API function *SendMessage*. The class *FSMSystem* takes care of transporting the message over the TCP/IP network and placing it in the local mailbox assigned to the destination automata.

Since individual automata based on the FSM Library only need to communicate among themselves, implicit TCP/IP support is sufficient. The need to communicate with other program components that are not based on the

FSM Library, and that use TCP/IP sockets, directly leads to the requirement for explicit TCP/IP support. To fulfill that requirement, the FSM Library also provides explicit (in addition to implicit) TCP/IP support in a form of traditional TCP/IP socket abstraction. Of course, the automata instance that uses these additional API features must be aware and capable of handling details of TCP/IP communication (IP addresses and port numbers).

Explicit TCP/IP support is provided by two additional classes, namely, *FSMSystemWithTCP* and *NetFSM*. These two classes enable the FSM Library–based automata to directly communicate over the TCP/IP protocol stack with other FSM Library–based automata, or with other TCP/IP program components, e.g., a Web server or SIP client. As their names suggest, the class *FSMSystemWithTCP* is used instead of the class *FSMSystem*, and the class *NetFSM* is a logical counterpart of the class *FiniteStateMachine*.

6.6.1 Class *FSMSystemWithTCP*

The class *FSMSystemWithTCP* is derived from the class *FSMSystem* by extending it with support for communication over the TCP/IP family of protocols. It inherits the basic functionality of the base class, which has been described previously (see Section 6.2.1 describing the class *FSMSystem*).

In contrast to single-processor applications, distributed applications comprise parts (i.e., groups of automata) that are started independently. Because of this, two groups of automata executing on different processors must establish a TCP/IP connection at their startup. The connection establishing procedure is symmetric: This means that either side of the party—or both—must start their local TCP servers by calling the function *InitTCPServer*. The opposite side establishes the connection by calling the function *establishConnection*.

Example:

```
// In processor 1 (server)
//
// Initialize kernel.
fsmSystem1->InitKernel(buffClassNo,buffersCount,buffersLength,2);

// Initialize TCP/IP server on port number 5000.
// NetFSM_Automata1 is derived from NetFSM.
fsmSystem1->InitTCPServer(5000,NetFSM_Automata1);

// In processor 2 (client)
//
// Set server TCP/IP parameters (port, IP address).
// Establish the connection.
fsmSystem2.setPort(5000);
fsmSystem2.setIP("192.168.77.77");
fsmSystem2.establishConnection();
...
```

This example shows the code excerpts for the TCP/IP server and client machines, named processor 1 and processor 2. At startup, the server initializes

the FSM Library kernel by calling the function *InitKernel* (its parameters are the number of buffer types, their count, length, and the number of the mailboxes to be used). Next, it calls the function *InitTCPServer* to start the TCP/IP server. We assumed in this example that the class *NetFSM_Automata1* is derived from the class *NetFSM*.

Alternately, the client sets the TCP port number (5000) by calling the function *setPort* and the IP address of the TCP server (192.168.77.77) by calling the function *setIP*, and establishes the connection with the server by calling the function *establishConnection*.

6.6.2 Class *NetFSM*

The class *NetFSM* is derived from the base class *FiniteStateMachine* by extending its basic functionality with support for the communication over the TCP/IP infrastructure. The inherited basic functionality has been described previously (see Section 6.2.2 describing the class *FiniteStateMachine*). The basic functionality is extended with the abstraction enabling TCP/IP communication by adding three new function members. The new functions are the following:

```
virtual void convertFSMToNetMessage()=0;
virtual uint16 convertNetToFSMMessage()=0;
virtual uint8 getProtocolInfoCoding()=0;
```

These functions are used to convert the internal message format (abbreviated as *FSM*) into an external, or network message format (abbreviated as *Net*), and vice versa. Normally, automata executing in the same processor exchange internal messages coded in internal message format. However, this message format is not suitable for transmission over the network. Most commonly, the message must be serialized, i.e., transformed from the data object and structure form into an external message in accordance with a given external message format. This is a series of bits, sometimes grouped in octets or words, that are transmitted over the communication line.

The functions listed above are virtual functions and therefore the programmer must define them while they write a class that is derived from the class *NetFSM*. The message format conversion functions naturally read a message from some input buffer, convert it into a requested format, and write the output to an output buffer.

The function *convertFSMToNetMessage* is not intended to be used directly by the communicating automata, but rather to be called internally by the FSM Library kernel to convert an internal message into the external one before it can be sent over the network. Therefore, the input of this function is the internal message, and its output is the corresponding output message. The parameters of this function specify the pointer to the internal message *fsmMessageS*, its length *fsmMessageLength*, the pointer to the output, the

external message *protocolMessageS*, and its length *sendMsgLength*. The programmer must specify the mapping algorithm by writing this function.

Symmetrically, the function *convertNetToFSM* is intended to be used by the FSM library kernel to convert an external message received over the network into an internal message representation, which must be delivered to the local mailbox and processed further by the corresponding local automata. The input of this function is the external message and the output is the internal message. The parameters of this function specify the pointer to the external message *protocolMessageR*, its length *receivedMessageLength*, the pointer to the output, internal message *fsmMessageR*, and its length *fsmMessageRLength*.

The function *getProtocolInfoCoding* returns the code of the type of external information coding. An instance of the class *NetFSM*, referred to as *net automata*, initiates the transmission of the message across the TCP/IP network by calling the function *sentToTCP*. This function may throw an exception in the case of an error, e.g., when net automata wants to send a message after the TCP connection has been closed.

Example:

```
// PrepareNewMessage parameters: buffer size and message type
PrepareNewMessage(0xAA,MESSAGE_NAME);

// Fill in message header:
// destination automata type, its ID, and its group ID (if relevant)
SetMsgToAutomata(AUTOMATA_TYPE);
SetMsgObjectNumberTo(automataId);
SetMsgToGroup(INVALID_08);
// Add parameters.
AddParamByte(PARAM_1,byte);
AddParamWord(PARAM_2,word);
AddParam(PARAM_3,parameterLength,parameterPointer);

// Send message to local mailbox:
// SendMessage(AUTOMATA_MBX_ID);
// or send it over TCP/IP network:
sendToTCP();
```

The example above demonstrates how automata can prepare a message and send it over a TCP/IP network. The message is prepared like any other message. The function *PrepareNewMessage* is used to allocate a buffer for the message and to specify a message name. A series of already described functions is then used to fill in the message header and add the message parameters (see the second example in Section 6.5 describing message management). At the end, instead of sending the message to the local mailbox by calling the function *SendMessage*, the message is sent over the TCP/IP network by calling the function *sendToTCP*.

A net finite state machine receives the messages equally as simple automata (instances of the class *FiniteStateMachine*) do, just by reading its local mailbox.

6.7 Global Constants, Types, and Functions

The file *kernelConsts.h* defines the global constants, types, and functions used by the FSM Library kernel. The constants and their values are as follows:

```
MSG_FROM_AUTOMATA = 0; // Source automata ID (BYTE)
MSG_FROM_GROUP = 1; // Source automata group ID (BYTE)
MSG_TO_AUTOMATA = 2; // Destination automata ID (BYTE)
MSG_TO_GROUP = 3; // Destination automata group ID (BYTE)
MSG_CODE = 4; // Message code(WORD)
MSG_OBJECT_ID_FROM = 6; // Source automata instance ID (DWORD)
MSG_OBJECT_ID_TO = 10; // Destination automata ID (DWORD)
CALL_ID = 14; // Call (process) ID
MSG_INFO_CODING = 18; // Info coding type, 0 = StandardMessage
MSG_LENGTH = 19; // Message payload length
MSG_INFO = 21; // Message payload offset
MSG_HEADER_END = MSG_INFO; // End of message header

INVALID_08 = 0xff; // Mask for 8 bits
INVALID_16 = 0xfff; // Mask for 16 bits
INVALID_32 = 0xffffffff; // Mask for 32 bits
```

The global data types are as follows:

```
int8, uint8 // BYTE
int16, uint16 // WORD
int32, uint32 // DWORD
```

The utility functions provided for the load-store manipulation with various data types are as follows:

```
void SetUint16(uint8 *addr,uint16 value);
void SetUint32(uint8 *addr,uint32 value);
uint16 GetUint16(uint8 *addr);
uint32 GetUint32(uint8 *addr);
```

The utility functions are provided to avoid cast operators in C/C++ programs because some microcontrollers do not allow word or double-word memory access to odd memory addresses.

6.8 API Functions

The FSM Library API functions are grouped into the following eight groups:

- *FSMSystem* constructor (Table 6.2)
- *FSMSystem* member functions (Table 6.3)
- *FSMSystemWithTCP* constructor (Table 6.4)

TABLE 6.2

FSMSystem Constructor Summary

`FSMSystem(uint8 numOfAutomata, uint8 numberOfMbx)` The constructor initializes the object that represents the FSM system, along with the data structures needed for its proper operation.

TABLE 6.3

FSMSystem Member Functions Summary

Type	Member Function
`Void`	`Add (ptrFiniteStateMachine object, uint8 automataType, uint32 numOfObjects, bool useFreeList=false)` This function adds the first instance of each automata type to the FSM system.
`Void`	`Add(ptrFiniteStateMachine object, uint8 automataType)` This function adds all the automata instances of the given type to the FSM system, except for the first instance.
`Void`	`InitKernel(uint8 buffClassNo, uint32 *buffersCount, uint32 *buffersLength, uint8 numOfMbxs=0, TimerResolutionEnum timerRes=Timer1s)` This function initializes the elements of the kernel responsible for time, buffer, and message management.
`Void`	`Remove(uint8 automataType)` This function removes all the instances of the given automata type from the FSM system.
`ptrFiniteStateMachine`	`Remove(uint8 automataType, uint32 object)` This function removes the given instance of the given automata type.
`Virtual void`	`Start()` This function starts the FSM system.
`Void`	`StopSystem()` This function stops the FSM system.

TABLE 6.4

FSMSystemWithTCP Constructor Summary

`FSMSystemWithTCP(uint8 numOfAutomata, uint8 numberOfMbx)` The constructor initializes the object that represents the FSM system supporting communication over the TCP/IP network, along with the data structures needed for its proper operation.

- *FSMSystemWithTCP* member functions (Table 6.5)
- *FiniteStateMachine* constructor (Table 6.6)
- *FiniteStateMachine* member functions (Table 6.7)
- *NetFSM* constructor (Table 6.8)
- *NetFSM* member functions (Table 6.9)

The following sections contain a detailed description of *FSMSystem* library API functions.

TABLE 6.5

FSMSystemWithTCP Member Functions Summary

Type	Member Function
`int`	`InitTCPServer(uint16 port, uint8 automataType, char *ipAddress=0, unsigned char *parm=0, int length=0)` This function initializes the TCP server. Once initialized, the server waits for a request to establish the TCP connection with a remote client.

TABLE 6.6

FiniteStateMachine Constructor Summary

`FiniteStateMachine(uint16 numOfTimers=DEFAULT_TIMER_NO, uint16 numOfState=DEFAULT_STATE_NO, uint16 maxNumOfProceduresPerState=DEFAULT_PROCEDURE_NO_PE_STATE, bool getMemory=true)` This constructor initializes the object that represents the instance of a given automata type, along with the data structures needed for its proper operation.

TABLE 6.7

FiniteStateMachine Member Functions Summary

Type	Member Function
`uint8*`	`AddParam(uint16 paramCode, uint32 paramLength, uint8 *param)` This function is used to add a given parameter of the given length to the new message.
`uint8*`	`AddParamByte(uint16 paramCode, BYTE param)` This function is used to add the given parameter of length 1 byte to the new message.
`uint8*`	`AddParamDWord(uint16 paramCode, DWORD param)` This function is used to add the given parameter of length 4 bytes to the new message.
`uint8*`	`AddParamWord(uint16 paramCode, WORD param)` This function is used to add the given parameter of length 2 bytes to the new message.
`virtual void`	`CheckBufferSize(uint32 paramLength)` This function provides a new message buffer with a size sufficient enough to accept a parameter of the given length.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`virtual void`	`ClearMessage()` This function returns the buffer allocated for the current message to the kernel and assigns value *NULL* to the internal pointer to the current message. The current message is the last message received by the automata instance.
`virtual void`	`CopyMessage()` This function makes a copy of the current message and assigns that copy to the new message.
`virtual void`	`CopyMessage(uint8 *msg)` This function makes a copy of the given message and assigns that copy to the new message.
`virtual void`	`CopyMessageInfo(uint8 infoCoding, uint16 lengthCorrection=0)` This function copies the part of the message containing useful information, referred to as a payload (a message without its header), from the current to the new message.
`virtual void`	`Discard(uint8* buff)` This function deletes the message placed in the given buffer and returns the buffer to the kernel.
`void`	`DoNothing()` This function performs no operation. It is called when the automata receives an unexpected message, unless a new function is provided to handle unexpected messages.
`void`	`Free FSM()` This function reports to the FSM system that the automata instance has finished its current assignment and is free for further assignments.
`virtual uint8`	`GetAutomata()=0` This function returns the identification of the automata type for this automata instance.
`uint8`	`GetBitParamByteBasic(uint32 offset, uint32 mask=MASK_32_BIT)` This function returns the value of the current message parameter of length 1 byte masked with the given mask.
`uint16`	`GetBitParamWordBasic(uint32 offset, uint32 mask=MASK_32_BIT)` This function returns the value of the current message parameter of length 2 bytes masked with the given mask.
`uint32`	`GetBitParamDWordBasic(uint32 offset, uint32 mask=MASK_32_BIT)` This function returns the value of the current message parameter of length 4 bytes masked with the given mask.
`virtual uint8*`	`GetBuffer(uint32 length)` This function returns the buffer whose size is not less than the size given by the value of its parameter.
`uint32`	`GetBufferLength(uint8 *buff)` This function returns the size of the given buffer in bytes.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`virtual inline uint32`	`GetCallId()` This function returns the identification of the communication process in which this instance is currently involved, e.g., the call ID.
`uint32`	`GetCount(uint8 mbx)` This function returns the current number of messages in the given mailbox.
`virtual uint8`	`GetGroup()` This function returns the identification of the group of automata to which this instance belongs.
`virtual uint8`	`GetInitialState()` This function returns the identification of the initial state of this automata type.
`virtual inline uint8`	`GetLeftMbx()` This function returns the identification of the mailbox assigned to the automata instance that is logically to the left of this automata instance.
`virtual inline uint8`	`GetLeftAutomata()` This function returns the identification of the automata type that is logically to the left of this automata instance.
`virtual inline uint8`	`GetLeftGroup()` This function returns the identification of the group of automata that is logically to the left of this automata instance.
`virtual inline uint32`	`GetLeftObjectId()` This function returns the identification of the automata instance that is logically to the left of this automata instance.
`virtual uint8`	`GetMbxId()` This function returns the identification of the mailbox assigned to this automata instance.
`virtual MessageInterface*`	`GetMessageInterface(uint32 id)` This function returns the object that governs the coding of messages used by this automata instance. The returned object is an instance of the class derived from the class *Message Interface*.
`uint8*`	`GetMsg()` This function returns the first unread message from the mailbox assigned to this automata instance.
`static uint8*`	`GetMsg(uint8 mbx)` This function returns the first unread message from the mailbox identified by the value of its parameter.
`inline uint32`	`GetMsgCallId()` This function returns the identification of the communication process (e.g., call ID) from the current message.
`inline uint16`	`GetMsgCode()` This function returns the message code from the current message header.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`inline uint8`	`GetMsgFromAutomata()` This function returns the identification of the originating automata type from the current message.
`inline uint8`	`GetMsgFromGroup()` This function returns the identification of the group of the originating automata instance for the current message.
`inline uint8`	`GetMsgInfoCoding()` This function returns the identification of the information coding scheme used for the current message.
`inline uint16`	`GetMsgInfoLength()` This function returns the payload length of the current message in bytes.
`inline uint16`	`GetMsgInfoLength(uint8 *msg)` This function returns the payload length of the given message in bytes. The message is specified by its pointer.
`inline uint32`	`GetMsgObjectNumberFrom()` This function returns the identification of the originating automata instance from the current message.
`inline uint32`	`GetMsgObjectNumberTo()` This function returns the identification of the destination automata instance from the current message.
`inline uint8`	`GetMsgToAutomata()` This function returns the identification of the destination automata type from the current message.
`inline uint8`	`GetMsgToGroup()` This function returns the identification of the type of group of the destination automata from the current message.
`inline uint8*`	`GetNewMessage()` This function returns the address of the buffer that contains the new message.
`inline uint8`	`GetNewMsgInfoCoding()` This function returns the identification of the information coding scheme used for the new message.
`inline uint16`	`GetNewMsgInfoLength()` This function returns the payload length of the new message in bytes.
`uint8*`	`GetNextParam(uint16 paramCode)` This function returns the address of the next instance of the given type of message parameter within the current message.
`bool`	`GetNextParamByte(uint16 paramCode, BYTE &param)` This function searches for the next instance of the given type of the single-byte parameter in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`bool`	`GetNextParamDWord(uint16 paramCode, DWORD &param)` This function searches for the next instance of the given type of the 4-byte parameter in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*.
`bool`	`GetNextParamWord(uint16 paramCode, WORD &param)` This function searches for the next instance of the given type of the 2-byte parameter in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*.
`virtual uint32`	`GetObjectId()` This function returns the unique identification of this automata instance.
`uint8*`	`GetParam(uint16 paramCode)` This function returns the address of the first instance of the given type of the message parameter within the current message.
`bool`	`GetParamByte(uint16 paramCode, BYTE &param)` This function searches for the first instance of the given type of single-byte parameter in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*.
`bool`	`GetParamDWord(uint16 paramCode, DWORD &param)` This function searches for the first instance of the given type of 4-byte parameter in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*.
`bool`	`GetParamWord(uint16 paramCode, WORD &param)` This function searches for the first instance of the given type of 2-byte parameter in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*.
`PROC_FUN_PTR`	`GetProcedure(uint16 event)` This function returns the pointer to the event handler for the given event identifier and the current state of automata.
`virtual inline uint8`	`GetRightMbx()` This function returns the identification of the mailbox assigned to the automata instance that is logically to the right of this automata instance.
`virtual inline uint8`	`GetRightAutomata()` This function returns the identification of the automata type that is logically to the right of this automata instance.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`virtual inline uint8`	`GetRightGroup()` This function returns the identification of the type of the group of automata that is logically to the right of this automata instance.
`virtual inline uint32`	`GetRightObjectId();` This function returns the identification of the automata instance that is logically to the right of this automata instance.
`virtual inline uint8`	`GetState()` This function returns the identification of the current state of this automata instance.
`virtual bool`	`IsBufferSmall(uint8 *buff, uint32 length)` This function returns the value *true* if the size of the given buffer is not greater than the given size specified as the value of its second parameter; otherwise, it returns the value *false*.
`virtual void`	`Initialize()` This function defines the automata state transition event handlers and timers used by this automata type.
`void`	`InitEventProc(uint8 state, uint16 event, PROC_FUN_PTR fun)` This function defines the given state transition event handler for the given automata state and the given event (message code).
`void`	`InitTimerBlock(uint16 tmrId, uint32 count, uint16 signalId)` This function initializes the given timer by the given duration and the timer expiration message code.
`void`	`InitUnexpectedEventProc(uint8 state, PROC_FUN_PTR fun)` This function defines the given state transition event handler for unexpected events in the given automata state.
`bool`	`IsTimerRunning(uint16 id)` This function returns the value *true* if the given timer is active (running); otherwise, it returns the value *false*.
`void`	`NoFreeObjectProcedure(uint8 *msg)` This function defines the behavior of this automata type if the list of free automata of this type is used and if it is empty at the moment when a free instance is requested.
`virtual void`	`NoFreeInstances()` This function defines the behavior of the FSM system if a list of free automata is used and if it is empty at the moment when a free instance is requested.
`virtual bool`	`ParseMessage(uint8 *msg)` This function checks if the given message is coded properly and, if it is, it becomes the current message (its pointer is assigned to the internal variable *CurrentMessage*).

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`virtual void`	`PrepareNewMessage(uint8 *msg)` This function defines the given buffer as the new message buffer by assigning the given pointer to the internal variable *NewMessage*. The buffer is used as a working area for the construction of the new message.
`virtual void`	`PrepareNewMessage(uint32 length, uint16 code, uint8 infoCode = LOCAL_PARAM_CODING)` This function creates a new message of the given length with the given message code and the given type of information coding.
`virtual void`	`Process(uint8 *msg)` This function performs the preparations for the message processing and selects the state transition event handler based on the message code and current state of this automata instance.
`void`	`PurgeMailBox()` This function purges all the messages from the mailbox assigned to this automata type and releases all the buffers assigned to the messages.
`bool`	`RemoveParam(uint16 paramCode)` This function removes the given type of message parameter from the new message.
`virtual void`	`Reset()` This function resets this automata instance by returning it to its initial state and by stopping all its active timers.
`void`	`ResetTimer(uint16 id)` This function resets the internal timer block object and returns the buffer allocated by the *StartTimer* primitive to the FSM Library kernel.
`void`	`RestartTimer(uint16 tmrId)` This function restarts the given timer. It is logically equivalent to a sequence of *StopTimer* and *StartTimer* primitives.
`virtual void`	`RetBuffer(uint8 *buff)` This function returns the given buffer to the FSM Library kernel. Normally, each memory buffer is returned at the end of its life cycle. Failure to do so leads to a memory leak problem.
`void`	`ReturnMsg(uint8 mbxId)` This function makes a copy of the current message and sends it to the given mailbox. This primitive is used frequently for message forwarding. On many occasions, the communication process must react in this simple way.
`void`	`SetBitParamByteBasic(BYTE param, uint32 offset, uint32 mask=MASK_32_BIT)` This function sets the given single byte parameter of the new message to the result of the bit-wise inclusive OR operation applied to the given parameter and its previous value masked (bit-wise AND operation) with the given bit-mask.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`void`	`SetBitParamDWordBasic(DWORD param, uint32 offset, uint32 mask=MASK_32_BIT)` This function sets the given 4-byte parameter of the new message to the result of the bit-wise inclusive OR operation applied to the given parameter and its previous value masked (bit-wise AND operation) with the given bit-mask.
`void`	`SetBitParamWord(WORD param, uint32 offset, uint32 mask=MASK_32_BIT)` This function sets the given 2-byte parameter of the new message to the result of the bit-wise inclusive OR operation applied to the given parameter and its previous value masked (bit-wise AND operation) with the given bit-mask.
`inline void`	`SetCallId()` This function sets the default value of the attribute *CallId* of this automata instance.
`inline void`	`SetCallId(uint32 id)` This function sets the given value of the attribute *CallId* of this automata instance.
`inline void`	`SetCallIdFromMsg()` This function sets the attribute *CallId* of this automata instance to the value of the parameter *CallId* of the current message. This primitive is used to store the reference number specific to the communication protocol.
`virtual void`	`SetDefaultFSMData()` This function sets the automata-specific data to their default values. It is typically used before the normal operation phase.
`virtual void`	`SetDefaultHeader(uint8 infoCoding)` This function sets the default header field values for the given type of the message information coding.
`inline void`	`SetGroup(uint8 id)` This function sets the identification of the group of automata for this automata type to the given value. This primitive is used to declare the group membership.
`virtual void`	`SetInitialState()` This function sets the current state of this automata instance to its initial state.
`static void`	`SetKernelObjects(TPostOffice *postOffice, TBuffers *buffers, CTimer *timer)` This function sets the *FSMSystem* library kernel objects (post office, buffers, and timers), which are common for all of the automata in the FSM system.
`inline void`	`SetLeftMbx(uint8 mbx)` This function sets the identification of the mailbox assigned to the automata instance that is logically to the left of this automata instance.
`inline void`	`SetLeftAutomata(uint8 automata)` This function sets the identification of the automata type that is logically to the left of this automata instance.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`inline void`	`SetLeftObject(uint8 group)` This function sets the identification of the type of the group of automata that is logically to the left of this automata instance.
`inline void`	`SetLeftObjectId(uint32 id)` This function sets the identification of the automata instance that is logically to the left of this automata instance.
`static void`	`SetLogInterface(LogInterface *logingObject)` This function defines the object responsible for message logging. The object is an instance of a class derived from the class *LogInterface*.
`inline void`	`SendMessage(uint8 mbxId)` This function sends a new message to the given mailbox. The mailbox is specified by its identification.
`inline void`	`SendMessage(uint8 mbxId, uint8 *msg)` This function sends the given message to the given mailbox.
`void`	`SetMessageFromData()` This function sets the header fields of the new message related to the originating automata instance to the values specific to this automata instance.
`inline void`	`SetMsgCallId(uint32 id)` This function sets the call ID parameter of the new message to the given value.
`inline void`	`SetMsgCallId(uint32 id, uint8 *msg)` This function sets the call ID parameter of the given message to the given value.
`inline void`	`SetMsgCode(uint16 code)` This function sets the message code parameter of the new message to the given value.
`inline void`	`SetMsgCode(uint16 code, uint8 *msg)` This function sets the message code parameter of the given message to the given value.
`inline void`	`SetMsgFromAutomata(uint8 from)` This function sets the type of the originating automata parameter of the new message to the given value.
`inline void`	`SetMsgFromAutomata(uint8 from, uint8 *msg)` This function sets the type of the originating automata parameter of the given message to the given value.
`inline void`	`SetMsgFromGroup(uint8 from)` This function sets the type of the originating group of automata parameters of the new message to the given value.
`inline void`	`SetMsgFromGroup(uint8 from, uint8 *msg)` This function sets the type of the originating group of automata parameters of the given message to the given value.
`inline void`	`SetMsgInfoCoding(uint8 codingType)` This function sets the message information coding parameter of the new message to the given value.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`inline void`	`SetMsgInfoCoding(uint8 codingType, uint8 *msg)` This function sets the message information coding parameter of the given message to the given value.
`inline void`	`SetMsgInfoLength(uint16 length)` This function sets the message payload (useful information) length parameter of the new message.
`inline void`	`SetMsgInfoLength(uint16 length, uint8 *msg)` This function sets the message payload (useful information) length parameter of the given message.
`inline void`	`SetMsgObjectNumberFrom(uint32 from)` This function sets the originating automata instance identification parameter of the new message to the given value.
`inline void`	`SetMsgObjectNumberFrom(uint32 from, uint8 *msg)` This function sets the originating automata instance identification parameter of the given message to the given value.
`inline void`	`SetMsgObjectNumberTo(uint32 to)` This function sets the destination automata instance identification parameter of the new message to the given value.
`inline void`	`SetMsgObjectNumberTo(uint32 to, uint8 *msg)` This function sets the destination automata instance identification parameter of the given message to the given value.
`inline void`	`SetMsgToAutomata(uint8 to)` This function sets the destination automata type identification parameter of the new message to the given value.
`inline void`	`SetMsgToAutomata(uint8 to, uint8 *msg)` This function sets the destination automata type identification parameter of the given message to the given value.
`inline void`	`SetMsgToGroup(uint8 to)` This function sets the destination automata group identification parameter of the new message to the given value.
`inline void`	`SetMsgToGroup(uint8 to, uint8 *msg)` This function sets the destination automata group identification parameter of the given message to the given value.
`void`	`SendMessageLeft()` This function sends the new message to the mailbox assigned to the automata instance that is logically to the left of this automata instance.
`void`	`SendMessageRight()` This function sends the new message to the mailbox assigned to the automata instance that is logically to the right of this automata instance.
`inline void`	`SetNewMessage(uint8 *msg)` This function sets the new message to the given message by assigning the given message pointer to the internal pointer to the new message.
`inline void`	`SetObjectId(uint32 id)` This function sets the identification of this automata instance to the given value.

(*Continued*)

TABLE 6.7 (CONTINUED)

FiniteStateMachine Member Functions Summary

Type	Member Function
`inline void`	`SetRightMbx(uint8 mbx)` This function sets the identification of the mailbox assigned to the automata instance that is logically to the right of this automata instance.
`inline void`	`SetRightAutomata(uint8 automata)` This function sets the identification of the automata type that is logically to the right of this automata instance.
`inline void`	`SetRightObject(uint8 group)` This function sets the identification of the type of the group of automata that is logically to the right of this automata instance.
`inline void`	`SetRightObjectId(uint32 id)` This function sets the identification of the automata instance that is logically to the right of this automata instance.
`inline void`	`SetState(uint8 state)` This function sets the identification of the current state of this automata instance.
`void`	`StartTimer(uint16 tmrId)` This function starts the given timer. The timer is specified by its identification.
`void`	`StopTimer(uint16 tmrId)` This function stops the given timer. The timer is specified by its identification.
`static void`	`SysClearLogFlag()` This function stops the logging of the messages exchanged by the automata.
`static void`	`SysStartAll()` This function starts the logging of the messages exchanged by the automata.

TABLE 6.8

NetFSM Constructor Summary

`NetFSM(uint16 numOfTimers=DEFAULT_TIMER_NO, uint16 numOfState=DEFAULT_STATE_NO, uint16 maxNumOfProceduresPerState=DEFAULT_PROCEDURE_NO_PER_STATE, bool getMemory=true)` The constructor initializes the object that represents an instance of the given automata type, along with the data structures needed for its proper operation.

TABLE 6.9

NetFSM Member Functions Summary

Type	Member Function
`virtual void`	`convertFSMToNetMessage()` This function converts the internal message format into the external message format appropriate for the transmission over the TCP/IP network.
`virtual uint16`	`convertNetToFSMMessage()` This function converts the external message format into the internal message format appropriate for communication within the FSM system.
`void`	`establishConnection()` This function establishes the TCP connection between two geographically distributed FSM systems.
`virtual uint8`	`getProtocolInfoCoding()` This function returns the identification of the type of the external message coding.
`void`	`sendToTCP()` This function sends the new message to the remote FSM system over the previously established TCP connection.

6.8.1 *FSMSystem*

Function prototype:

```
FSMSystem(
 uint8 numOfAutomata,
 uint8 numberOfMbx)
```

Function description: This constructor initializes the object that represents the FSM system together with the data structures needed for its proper operation.

Parameters:

numOfAutomata: the number of various automata types to be added to the FSM system

numberOfMbx: the number of mailboxes to be used by the FSM system

Note: Typically, a single mailbox is assigned to each automata type, but other arrangements are also allowed. Normally, an automata type corresponds to a protocol. For example, the IP protocol may be implemented as one automata type, and the TCP protocol may be implemented as another automata type. A typical arrangement would be to assign one mailbox to IP and one to TCP. Another arrangement would be to assign two mailboxes to each protocol. For example, in this arrangement, IP would use the first mailbox to receive the messages from network interfaces (drivers) and the second mailbox to receive the messages from TCP. Yet another arrangement would

be to assign a single mailbox to all the protocols. Finally, a set of mailboxes can be used to prioritize the messages. For example, three mailboxes may be used to distinguish high, middle, and low priority messages.

6.8.2 Add(*ptrFiniteStateMachine, uint8, uint32, bool*)

Function prototype:

```
void Add(
 ptrFiniteStateMachine object,
 uint8 automataType,
 uint32 numberOfObjects,
 bool useFreeList = false)
```

Function description: This function adds the first instance of each automata type to the FSM system. At the same time, this function defines the unique identification of this automata type and the number of instances of this automata type that will be subsequently added to the FSM system. It also declares a group of instances of this automata type as either a set of resources to be used individually or as a pool of resources of the same type available for dynamic allocation.

Function parameters:

object: the pointer to the first instance of this automata type to be added to the FSM system

automataType: the unique identification of this type of automata

numberOfObjects: the total number of instances of this type to be added to the FSM system

useFreeList: the indicator selecting the mode of usage of individual instances of this type

Note: Typically, the FSM system is created at system startup, and then groups of various automata types are added to it. As a rule, the first instance of the given automata type is added by this function. Its parameters specify, in order from left to right, the pointer to the first object of this type, the identification of this automata type, the total number of instances that will be added to the FSM system, and the mode of individual instance allocation. This last parameter has a default value *false*, which means that each automata instance represents an individual resource. If this default is overridden by the value *true*, the group of instances of this automata type represents a pool of resources of the same type. The individual instances from this pool are allocated dynamically and on-demand, based on the use of the internal FSMSystem library kernel list of resources of the given type. (This is the origin of the name of the last parameter of this function, *useFreeList*.) This dynamic allocation is requested by sending a message to an unknown

automata, which is identified by the instance identification set to the value –1 (see function *SetMsgObjectNumberTo*).

6.8.3 Add(*ptrFiniteStateMachine, uint8*)

Function prototype:

```
void Add(
 ptrFiniteStateMachine object,
 uint8 automataType)
```

Function description: This function adds all the automata instances except the first instance of the given type to the FSM system. It assumes that the first instance of this automata type has been added previously to the FSM system by calling the overloaded function *Add* with four parameters in its signature.

Function parameters:

object: the pointer to the instance of this automata type to be added to the FSM system

automataType: the unique identification of this automata type

Note: As already mentioned, after the FSM system is created at system startup, the groups of various automata types are added to it. As a rule, the first instance of the given automata type is added by the overloaded function *Add* with four parameters in its signature (see the previous section for more details on its parameters). All the other instances of the given automata type are added to the FSM system by this overloaded function *Add*. An advantage of differentiating these two functions becomes obvious in a dynamic environment where objects are created on-demand and added to the FSM system. If the given automata type already exists, and a need arises for another instance of it, this overloaded *Add* function is sufficient.

6.8.4 *InitKernel*

Function prototype:

```
void InitKernel(
 uint8 buffClassNo,
 uint32 *buffersCount,
 uint32 *buffersLength,
 uint8 numOfMbxs=0,
 TimerResolutionEnum timerRes = Timer1s)
```

Function description: This function initializes the elements of the kernel responsible for time, buffer, and message management. The parameters of this function specify the number of buffer types, the number of instances per buffer type and their lengths, the number of mailboxes to be used by

the automata added to the FSM system, and the basic timer resolution. The default value of the basic timer resolution is 1 sec, which is defined by the symbolic constant *Timer1s*.

Function parameters:

buffClassNo: the number of buffer types

buffersCount: the pointer to the array of the numbers of instances of the corresponding buffer types

buffersLength: the pointer to the array of the sizes of the corresponding buffer types

numOfMbxs: the number of the mailboxes

timerRes: the basic timer resolution

Note: This function essentially initializes the *FSMSystem* library kernel. It must be called after the FSM system has been created and before it can be started. It also assumes that the arrays of the cardinal numbers and the sizes of individual buffer types were already created and filled by the programmer. Because the specification of the buffers to be provided by the kernel may look cumbersome, we provide the following example. Suppose that a need arises for three buffer types, namely, small, medium, and large. The programmer should set the first parameter of this function to the number 3. Next, suppose that the programmer needs 300 small buffers, 200 medium buffers, and 100 large buffers, and that their sizes should be 64, 128, and 256 bytes, respectively. Before calling this function, the programmer should create the following two arrays:

- Array of cardinal numbers = [300, 200, 100]
- Array of sizes = [64, 128, 256]

Finally, the programmer should specify the pointers to these two arrays as the second and the third parameter of this function.

6.8.5 *Remove(uint8)*

Function prototype:

```
void Remove(unit8 automataType)
```

Function description: This function removes all instances of the given automata type from the FSM system.

Function parameters:

automataType: the type of automata to be removed from the system

Note: First, the FSM system removes all instances of the given automata type from the FSM system. Next, the kernel frees all the memory zones occupied by the internal data structures used by the automata of this type.

6.8.6 *Remove*(*uint8*, *uint32*)

Function prototype:

```
ptrFiniteStateMachine Remove(
 uint8 automataType
 uint32 object)
```

Function description: This function removes the given instance of the given automata type. The parameters of this function specify the identification of the automata type and the identification of the automata instance.

Function parameters:

automataType: the identification of the automata type

object: the identification of the instance of the given automata type

Function returns: This function returns the pointer to the automata instance removed from the FSM system.

6.8.7 *Start*

Function prototype:

```
virtual void Start()
```

Function description: This function starts the FSM system and is the main function of the FSM system. In this function, the FSM system thread enters a loop in which it reads the kernel mailboxes and distributes the messages to the destination automata.

Note: The FSM system thread remains in the loop while the internal attribute *SystemWorking* is set to the value *true*. A typical implementation of the FSM system thread is shown in the example in Section 6.2.1.2.

6.8.8 *StopSystem*

Function prototype:

```
void StopSystem()
```

Function description: This function stops the FSM system. It sets the internal attribute *SystemWorking* to the value *false*, thus causing the FSM system thread to exit its loop and stop the FSM system.

Note: If the function *Start* has been called from the separate operating system thread, the call to the function *StopSystem* will cause the termination of that thread.

6.8.9 *FSMSystemWithTCP*

Function prototype:

```
FSMSystemWithTCP(
 uint8 numOfAutomata,
 uint8 numberOfMbx)
```

Function description: This constructor initializes the object that represents the FSM system supporting communication over TCP/IP network, along with the data structures needed for its proper operation. Its parameters specify the number of automata types to be added to the FSM system and the number of mailboxes.

Function parameters:

numOfAutomata: the number of automata types that will be added to the FSM system

numberOfMbx: the number of mailboxes that will be used by the automata added to the FSM system

Note: Typically, a single mailbox is assigned to each automata type included in the FSM system, but other arrangements are also allowed. For example, a single mailbox may be assigned to all the automata types included in the FSM system. Also allowed is to assign an arbitrary number of mailboxes to each automata type, e.g., to enable message prioritization.

6.8.10 *InitTCPServer*

Function prototype:

```
int InitTCPServer(
 uint16 port,
 unit8 automataType,
 char *ipAddress = 0,
 unsigned char *parm = 0,
 int length = 0)
```

Function description: This function initializes the TCP server. Once initialized, the server waits for a request to establish the TCP connection with a remote client. The parameters of this function specify the number of the TCP port on which the server awaits the connection request, the automata type included in the FSM system engaged in the communication, the server IP address, the pointer to the area where the connection

parameters should be passed to the specified automata type, and the parameter lengths in bytes. After reception of the request, the server allocates an instance of the given automata type and passes the connection together with the received parameters to the allocated automata instance. Further communication continues directly between the remote client and the allocated automata instance, i.e., the server is completely isolated from it.

Function parameters:

Port: the number of the TCP port on which the server awaits a connection request

automataType: the automata type included in the FSM system that is engaged in the communication. This automata type must be derived from the class *NetFSM*. After the connection has been initially established, the server transfers it to the allocated instance of this automata type.

ipAddress: the pointer to the server IP address

parm: the pointer to the area where the parameters received while establishing the connection should be passed and subsequently taken by to the specified automata type

length: the parameter lengths specified by the previous pointer, in bytes

Function returns: If the TCP server awaiting a request from a remote client is successfully started, this function returns the value 0. Otherwise, it returns the value –1.

Note: This function should be called only once, just initially to start the TCP server.

6.8.11 *FiniteStateMachine*

Function prototype:

```
FiniteStateMachine(
 unit16 numOfTimers = DEFAULT_TIMER_NO,
 uint16 numOfState = DEFAULT_STATE_NO,
 uint16 maxNumOfProceduresPerState = DEFAULT_PROCEDURE_NO_PER_STATE,
 bool getMemory = true)
```

Function description: This constructor initializes the object that represents the instance of a given automata type together with the data structures needed for its proper operation. Its parameters specify the number of timers to be used by this automata type, the number of the states that this automata type has, the maximal number of state transitions per state, and the indicator specifying whether this constructor should reserve the memory for the objects that represent the states and state transitions of this automata type

or not. The default value of this indicator is *true*, which means that this constructor is responsible for memory allocation.

Function parameters:

numOfTimers: the number of the timers to be used by this automata type

numOfState: the number of the states that this automata type has

maxNumOfProceduresPerState: the maximal number of state transitions per state

getMemory: the memory allocation indicator (by default, its value is *true*)

Note: This constructor may be called either with some or without any of the parameters. If the parameter is not specified, the constructor will use its default value. The indicator *getMemory* may be set to the value *false* when the programmer wants to do manual memory allocation to optimize overall memory consumption.

6.8.12 *AddParam*

Function prototype:

```
uint8 *AddParam(
 uint16 paramCode,
 uint32 paramLength,
 uint8 *param)
```

Function description: This function is used to add a given parameter of a given length to the new message. The parameters of this function specify the unique identification of the parameter type, the parameter length in bytes, and the pointer to the parameter itself. If the parameter to be added to the message is too large to fit in the buffer that is assigned to the new message, this function will get a bigger buffer, copy the new message into it, add the parameter, and release the old buffer.

Function parameters:

paramCode: the parameter type

paramLength: the parameter length, in bytes

param: the pointer to the parameter

Function returns: This function returns the pointer to the buffer that contains the new message.

Note: This function enables the programmer to add a parameter of an arbitrary size to the new message with the limitation that it must not exceed the maximal parameter length specified for the given type of message coding

(e.g., for the type *StandardMessage*, the maximal parameter length is 256 bytes). The message parameters in *StandardMessage* are sorted by ascending order of their corresponding type identifiers.

6.8.13 *AddParamByte*

Function prototype:

```
uint8 *AddParamByte(
 uint16 paramCode,
 BYTE param)
```

Function description: This function is used to add the given parameter of length 1 byte to the new message. The parameters of this function specify the unique identification of the parameter type and the parameter value.

Function parameters:

paramCode: the parameter type

param: the parameter value

Function returns: This function returns the pointer to the buffer that contains the new message.

Note: The total message length must not exceed the limit specified for the given type of message coding. In any case, it must not exceed 8G bytes.

6.8.14 *AddParamDWord*

Function prototype:

```
uint8 *AddParamDWord(
 uint16 paramCode,
 DWORD param)
```

Function description: This function is used to add the given parameter of length 4 bytes to the new message. The parameters of this function specify the unique identification of the parameter type and the parameter value.

Function parameters:

paramCode: the parameter type

param: the parameter value

Function returns: This function returns the pointer to the buffer that contains the new message.

Note: The total message length must not exceed the limit specified for the given type of message coding. In any case, it must not exceed 232 bytes.

6.8.15 *AddParamWord*

Function prototype:

```
uint8 *AddParamDWord(
 uint16 paramCode,
 WORD param)
```

Function description: This function is used to add the given parameter of length 2 bytes to the new message. The parameters of this function specify the unique identification of the parameter type and the parameter value.

Function parameters:

paramCode: the parameter type

param: the parameter value

Function returns: This function returns the pointer to the buffer that contains the new message.

Note: The total message length must not exceed the limit specified for the given type of message coding. In any case, it must not exceed 8G bytes.

6.8.16 *CheckBufferSize*

Function prototype:

```
uint8 *CheckBufferSize(uint32 paramLength)
```

Function description: This function provides a new message buffer with the size sufficient enough to accept the parameter of the given length. The parameter of this function specifies the parameter length in bytes.

Function parameters:

paramLength: the parameter length

Function returns: This function returns the pointer to the new message.

Note: This function is obsolete. In the previous version of the FSM Library, this function ensured the new message buffer management was transparent for the programmer. Typically, the programmer would call this function before calling some of the *AddParam* functions to ensure that the new message is stored in a buffer of sufficient size. This means that the buffer is large enough to accept a new parameter of the given size, in addition to the current content of the new message. Behind the scenes, this function checked the current size of the new message. If it was not sufficient, the function allocated a new, larger buffer; copied the current new message into it; released

the old buffer; and returned the pointer to the newly allocated buffer containing the new message. In the current version of the FSM Library, all the *AddParam* functions call this function internally at their very beginning, and the programmer no longer needs to call it explicitly.

6.8.17 *ClearMessage*

Function prototype:

```
virtual void ClearMessage()
```

Function description: This function returns the buffer allocated for the current message to the kernel and assigns the value *NULL* to the internal pointer to the current message. The current message is the last message received by the automata instance.

Note: If the *FSMSystem* library has been compiled for the debug mode, this function will additionally verify that the return value of the function is *NULL*.

6.8.18 *CopyMessage( )*

Function prototype:

```
virtual void CopyMessage()
```

Function description: This function makes a copy of the current message and assigns that copy to the new message. By definition, a current message is the last received message, and a new message is the message under construction to be subsequently sent. The value of the pointer to the current message copy is assigned to the internal pointer to the new message.

Note: This function first checks if the new message already exists by checking the internal pointer to the new message. If the new message has already been defined or is under construction (the internal pointer is not equal to the value *NULL*), the function releases the buffer that contains the new message and assigns the value *NULL* to the internal pointer. Next, the function makes a copy of the current message and assigns its address to the pointer to the new message. This function is typically used for message forwarding. The protocol *A* sends a message to the protocol *B*, which, in turn, forwards the copy of the same message to the protocol *C*.

6.8.19 *CopyMessage(uint*)*

Function prototype:

```
virtual void CopyMessage(uint8 *msg)
```

Function description: This function makes a copy of the given message and assigns that copy to the new message. The parameter of this function specifies the pointer to the original message.

Function parameters:

msg: the pointer to the original message

Note: This function assumes that the new message does not exist, i.e., the internal pointer to the new message should contain the value *NULL* before this function is called. However, if the new message already exists, this function will return its buffer and get a fresh buffer for the new message before copying the given message into it.

6.8.20 *CopyMessageInfo*

Function prototype:

```
virtual void CopyMessageInfo(
 uint8 infoCoding,
 uint16 lengthCorrection = 0)
```

Function description: This function copies the part of the message containing the useful information, referred to as a payload (message without its header), from the current message into the new message stored in a newly allocated buffer. The parameters of this function specify the type of information coding that governs the formatting and length correction of the message.

Function parameters:

infoCoding: the identification of the type of information coding

lengthCorrection: the message length correction

Note: The message length correction depends on the type of applied information coding. If the new message buffer does not exist, this function will get a buffer, assign it to the new message, and make the required copy.

6.8.21 *Discard*

Function prototype:

```
virtual void Discard(uint8* buff)
```

Function description: This function deletes the message placed in the given buffer and returns the buffer to the kernel. The parameter of this function specifies the buffer to be cleared and released.

Function parameters:

buff: the pointer to the buffer

6.8.22 *DoNothing*

Function prototype:

```
void DoNothing()
```

Function description: This function performs no operation. It is called when the automata receives an unexpected message, unless a new function to handle unexpected messages is defined. By definition, an unexpected message is any type of message that has not been defined as a legal type of message in the current automata state.

Note: This function may be redefined by calling the function *Init UnexpectedEventProc*, if a need exists for concrete functionality handling unexpected messages.

6.8.23 *FreeFSM*

Function prototype:

```
void FreeFSM()
```

Function description: This function reports to the FSM system that an automata instance has finished its current assignment and is free for further assignments. If the first instance of this automata type has been added to the FSM system with the parameter *useFreeList* set to the value *true*, the group of instances of this automata type is viewed as a pool of resources. In that case, this function returns the resource to the corresponding pool by queuing it to the internal list of the resources of the same type.

Note: If a group of instances of this automata type is used as a set of individual resources, rather than as a pool of resources (the parameter *useFree List* has been set to the value *false* when the first automata instance has been added to the FSM system), this function has no effect.

6.8.24 *GetAutomata*

Function prototype:

```
virtual uint8 GetAutomata() = 0
```

Function description: This function returns the identification of the automata type for this automata instance.

Function returns: This function returns the unique ID of the automata type.

Note: This function is a pure virtual function, which means that it must be defined in the class that models some concrete automata type. Typically, this function returns the constant value that represents the required identification. It finds this constant by looking up the table of identifications created by reading the file of all the known automata types at the FSM system startup time.

6.8.25 *GetBitParamByteBasic*

Function prototype:

```
unit8 GetBitParamByteBasic(
 uint32 offset,
 uint32 mask=MASK_32_BIT)
```

Function description: This function returns the value of the current message parameter of length 1 byte masked with the given mask. The parameters of this function specify the offset of the original parameter of the message and the value of the mask.

Function parameters:

offset: the offset of the original parameter of the message

mask: the value of the mask

Function returns: This function returns the result of the bit-wise AND operation between the value of the message parameter at the given message *offset* and the given value of the parameter *mask*.

Note: Normally, depending on the value of the parameter mask, testing the value of a single bit, or of a group of bits simultaneously, is possible in the parameter of size 1 byte that is at a given distance from the beginning of the message.

6.8.26 *GetBitParamWordBasic*

Function prototype:

```
unit8 GetBitParamWordBasic(
 uint32 offset,
 uint32 mask=MASK_32_BIT)
```

Function description: This function returns the value of the current message parameter of length 2 bytes masked with the given mask. The parameters of this function specify the offset of the original parameter of the message and the value of the mask.

Function parameters:

offset: the offset of the original parameter of the message

mask: the value of the mask

Function returns: This function returns the result of the bit-wise AND operation between the value of the message parameter at the given message *offset* and the given value of the parameter *mask*.

Note: Normally, depending on the value of the parameter mask, testing the value of a single bit, or a group of bits simultaneously, is possible in the parameter of size 2 bytes that is at a given distance from the beginning of the message.

6.8.27 *GetBitParamDWordBasic*

Function prototype:

```
unit8 GetBitParamDWordBasic(
 uint32 offset,
 uint32 mask=MASK_32_BIT)
```

Function description: This function returns the value of the current message parameter of length 4 bytes masked with the given mask. The parameters of this function specify the offset of the original parameter of the message and the value of the mask.

Function parameters:

offset: the offset of the original parameter of the message

mask: the value of the mask

Function returns: This function returns the result of the bit-wise AND operation between the value of the message parameter at the given message *offset* and the given value of the parameter *mask*.

Note: Normally, depending on the value of the parameter mask, testing the value of a single bit, or of a group of bits simultaneously, is possible in the parameter of size 4 bytes that is at a given distance from the beginning of the message.

6.8.28 *GetBuffer*

Function prototype:

```
virtual uint8 *GetBuffer(uint32 length)
```

Function description: This function returns a buffer whose size is not less than the size given by the value of its parameter. The parameter of this message specifies the minimal buffer length in bytes.

Function parameters:

length: the buffer length

Function returns: This function returns the pointer to a newly allocated buffer.

Note: The *FSMSystem* library kernel handles a limited number of buffer types with a limited number of instances per each type defined during the kernel initialization by calling the function *InitKernel*. By definition, this function first searches for the buffer types of the size that ideally match the desired buffer. If such a type does not exist, the function searches for the next size buffer type (in the increasing order of size). This allocation policy may yield a buffer of a size much bigger than needed, and the frequent occurrence of this type of allocation may lead to inefficient memory usage. For example, suppose that the programmer has mistakenly defined only two buffer sizes, small and large, such that not a single protocol message can fit into the small buffer. In this case, only the large buffers will be consumed, and the small buffers will not be used at all. Therefore, special care must be taken when defining the buffers before calling the function *InitKernel*.

Now, let us go back to the buffer allocation algorithm. When this function finds a buffer type of a sufficient size, it checks for a free buffer of that type. If no such type is found, the system is badly designed and a new buffer type must be added to the system. If such a buffer type exists, but no free buffers of that type are available, the function will look for the next size buffer. If all the buffers of the sufficient size are already allocated, the FSM system experiences a memory exhaustion problem. In the academic examples, the system is allowed to crash under these circumstances. However, industrial-strength applications require implementation of additional mechanisms, such as overload protection and intelligent automatic restarts.

6.8.29 *GetBufferLength*

Function prototype:

```
uint32 GetBufferLength(uint8 *buff)
```

Function description: This function returns the size of the given buffer in bytes. The parameter of this function specifies the pointer to the buffer.

Function parameters:

buff: the address of the buffer

Function returns: This function returns the specified buffer length in bytes.

6.8.30 *GetCallId*

Function prototype:

```
virtual inline uint32 GetCallId()
```

Function description: This function returns the identification of the communication process that this instance is currently involved in, e.g., the call ID. The actual meaning of this identification is application specific.

Function returns: This function returns the value of the attribute *CallId*.

Note: Historically, the attribute *CallId* is tied to call processing (e.g., Q.71) and signaling (e.g., SS7, DSS1) protocols, but it has also proved to be useful in modern multimedia protocols (e.g., H.323 and SIP). Generally, this attribute may be used as an identification of the process or transaction that engages more cooperative automata. If a single attribute is not sufficient, the programmer may introduce additional attributes in classes derived from the base class *FiniteStateMachine*.

6.8.31 *GetCount*

Function prototype:

```
uint32 GetCount(uint8 mbx)
```

Function description: This function returns the current number of messages in the given mailbox. The parameter of this message specifies the identification of the mailbox.

Function parameters:

mbx: the mailbox identification

Function returns: This function returns the number of unread messages contained in the mailbox of interest.

6.8.32 *GetGroup*

Function prototype:

```
virtual uint8 GetGroup()
```

Function description: This function returns the identification of the group of automata to which this instance belongs.

Function returns: This function returns a number that uniquely identifies the group of automata which, besides other members, includes this automata instance.

6.8.33 *GetInitialState*

Function prototype:

```
virtual uint8 GetInitialState()
```

Function description: This function returns the identification of the initial state of this automata type.

Function returns: This function returns the number that uniquely identifies the initial state of this automata type.

Note: The default value of the initial state is 0.

6.8.34 *GetLeftMbx*

Function prototype:

```
virtual inline uint8 GetLeftMbx()
```

Function description: This function returns the identification of the default mailbox assigned to the automata instance that is logically to the left of this automata instance.

Function returns: This function returns the number that uniquely identifies the default mailbox assigned to the left automata instance.

Note: Historically, the terms *left* and *right* automata instance originate from SDL, where an automata instance typically communicates with its left and right neighbors. These neighbors might have their own mailboxes, sometimes briefly called left and right mailboxes.

6.8.35 *GetLeftAutomata*

Function prototype:

```
virtual inline uint8 GetLeftAutomata()
```

Function description: This function returns the identification of the automata type that is logically to the left of this automata instance.

Function returns: This function returns the number that uniquely identifies the left automata type.

Note: By definition, left automata are logically placed to the left of the currently observed automata instance.

6.8.36 *GetLeftGroup*

Function prototype:

```
virtual linline uint8 GetLeftGroup()
```

Function description: This function returns the identification of the group of automata that is logically to the left of this automata instance.

Function returns: This function returns the number that uniquely identifies the left group of automata.

Note: By definition, a left group of automata is a group that contains left automata.

6.8.37 *GetLeftObjectId*

Function prototype:

```
virtual inline uint32 GetLeftObjectId()
```

Function description: This function returns the identification of the automata instance that is logically to the left of this automata instance.

Function returns: This function returns the number that uniquely identifies the left automata instance.

Note: By definition, left automata are logically placed to the left of the currently observed automata instance. This function returns the identification of the particular left automata instance with which the currently observed automata instance communicates.

6.8.38 *GetMbxId*

Function prototype:

```
virtual uint8 GetMbxId()
```

Function description: This function returns the identification of the default mailbox assigned to this automata type. Note that an instance of a given automata type may receive its messages through any mailbox, i.e., through the default mailbox as well as through other mailboxes. Alternately, a single mailbox may be assigned to more than one automata type.

Function returns: This function returns the number that uniquely identifies the default mailbox assigned to this automata instance.

Note: This function is a pure virtual function, which means that it must be defined by the programmer when they write a class derived from the class *FiniteStateMachine*. Typically, this function returns the constant value that represents the required mailbox identification (the content of the corresponding class field). This constant can be initially determined by looking up the table of identifications, and set by calling the function *SetMbxId*. The table of identifications can be created by reading the file containing all the known automata types at the FSM system startup time. A mailbox ID is typically a record field that describes a single automata type.

6.8.39 *GetMessageInterface*

Function prototype:

```
virtual MessageInterface *GetMessageInterface(uint32 id) = 0
```

Function description: This function returns the object that governs the coding of messages used by this automata instance. The parameter of this function specifies the identification of the information coding scheme. The returned object is an instance of the class derived from the class *MessageInterface*.

Function parameters:

id: the information coding scheme

Function returns: This function returns the pointer to the object responsible for parsing and coding the messages used by this automata instance.

Note: This function is a virtual function, which means that it must be defined when the programmer writes a class derived from the class *FiniteStateMachine*. The identification with the value 0 is reserved for the information coding used by the format of the class *StandardMessage*, which is a basic type of message supported by the *FSMSystem* library.

6.8.40 *GetMsg()*

Function prototype:

```
uint8* GetMsg()
```

Function description: This function returns the first unread message from the mailbox assigned to this automata instance.

Function returns: This function returns a pointer to the buffer that has been removed from the head of the list, which is hidden by the abstraction of the mailbox assigned to this automata instance. If no such buffer exists, i.e., if the list is empty, the function returns the value *NULL*.

6.8.41 *GetMsg(uint8)*

Function prototype:

```
static uint8* GetMsg(uint8 mbx)
```

Function description: This function returns the first unread message from the given mailbox. The parameter of this function specifies the identification of the mailbox.

Function parameters:

mbx: the mailbox ID

Function returns: This function returns the pointer to the buffer that has been removed from the head of the list, which is hidden by the abstraction of the given mailbox. If no such buffer exists, i.e., if the list is empty, the function returns the value *NULL*.

Note: Although this function is defined as a static function, a call to this function is not allowed before the kernel initialization and the FSM system startup. The call to this function made before that may cause unpredictable behavior.

6.8.42 *GetMsgCallId*

Function prototype:

```
inline uint32 GetMsgCallId()
```

Function description: This function returns the identification of the communication process (e.g., call ID) from the current message.

Function returns: This function returns the value of the attribute *CallId*.

Note: The attribute *CallId* is application specific. It can be used to indicate a process or a transaction in which more cooperating automata are involved. The size of *CallId* is 32 bits. It is considered large enough for most of the applications. To increase the size of *CallId*, the programmer would need to modify the base class *FiniteStateMachine*.

6.8.43 *GetMsgCode*

Function prototype:

```
inline uint16 GetMsgCode()
```

Function description: This function returns the message code from the current message header.

Function returns: This function returns the value of the message code from the header of the current (last received) message.

6.8.44 *GetMsgFromAutomata*

Function prototype:

```
inline uint8 GetMsgFromAutomata()
```

Function description: This function returns the identification of the originating automata type from the current message. This value is provided from the header of the current message.

Function returns: This function returns the value of the identification of the automata type that has created and sent the current message to this automata instance.

6.8.45 *GetMsgFromGroup*

Function prototype:

```
inline uint8 GetMsgFromGroup()
```

Function description: This function returns the identification of the group of the originating automata instance for the current message. This value is provided from the header of the current message.

Function returns: This function returns the value of the identification of the group of automata instance that has created and sent the current message to this automata instance.

6.8.46 *GetMsgInfoCoding*

Function prototype:

```
inline uint8 GetMsgInfoCoding()
```

Function description: This function returns the identification of the information coding scheme used for the current message.

Function returns: This function returns the value that identifies the type of information coding that has been used to create the current message.

Note: This information is provided from the header of the current message.

6.8.47 *GetMsgInfoLength()*

Function prototype:

```
inline uint16 GetMsgInfoLength()
```

Function description: This function returns the payload length of the current message in bytes.

Function returns: This function returns the value of the current message payload size in bytes.

Note: The length of the message header is not included in the length returned by this message. By definition, the total message length is the sum of the length of the message header and the length of the message payload.

6.8.48 *GetMsgInfoLength(uint8*)*

Function prototype:

```
inline uint16 GetMsgInfoLength(uint8 *msg)
```

Function description: This function returns the payload length of the given message in bytes. The parameter of this function specifies the pointer to the message.

Function parameters:

msg: the pointer to the message

Function returns: This function returns the value of the size of the given message payload in bytes.

Note: The length of the message header is not included in the length returned by this message. By definition, the total message length is the sum of the length of the message header and the length of the message payload.

6.8.49 *GetMsgObjectNumberFrom*

Function prototype:

```
inline uint32 GetMsgObjectNumberFrom()
```

Function description: This function returns the identification of the originating automata instance from the current message.

Function returns: This function returns the value that identifies the automata instance that has created and sent the message.

Note: This value is provided from the header of the current (last received) message.

6.8.50 *GetMsgObjectNumberTo*

Function prototype:

```
inline uint32 GetMsgObjectNumberTo()
```

Function description: This function returns the identification of the destination automata instance from the current message. This value is actually this automata instance.

Function returns: This function returns the value that identifies the automata instance that has received the message and that must process it.

Note: This value is provided from the header of the current (last received) message.

6.8.51 *GetMsgToAutomata*

Function prototype:

```
inline uint8 GetMsgToAutomata()
```

Function description: This function returns the identification of the destination automata type from the current message. This value is actually this automata type.

Function returns: This function returns the value that identifies the automata type that should receive the message and that should process it.

Note: This value is provided from the header of the current (last received) message.

6.8.52 *GetMsgToGroup*

Function prototype:

```
inline uint8 GetMsgToGroup()
```

Function description: This function returns the identification of the type of the group of the destination automata from the current message. This value is actually the group to which this automata type belongs.

Function returns: This function returns the value that identifies the group of automata that has received the message and that must process it.

Note: This value is provided from the header of the current (last received) message.

6.8.53 *GetNewMessage*

Function prototype:

```
inline uint8 *GetNewMessage()
```

Function description: This function returns the address of the buffer that contains the new message.

Function returns: This function returns the pointer to the already defined new message or the message under construction.

Note: If the new message does not exist, this function returns the value *NULL*. This function assumes that the programmer has already allocated a buffer for the new message by previously calling the function *PrepareNewMessage* or calling the function *GetBuffer*.

6.8.54 *GetNewMsgInfoCoding*

Function prototype:

```
inline uint8 GetNewMsgInfoCoding()
```

Function description: This function returns the identification of the information coding scheme used for the new message.

Function returns: This function returns the value that uniquely identifies the type of information coding.

Note: This value is provided from the header of the new message.

6.8.55 *GetNewMsgInfoLength*

Function prototype:

```
inline uint16 GetNewMsgInfoLength()
```

Function description: This function returns the payload length of the new message in bytes.

Function returns: This function returns the value of the new message payload size in bytes.

Note: The length of the message header is not included in the length returned by this message. By definition, the total message length is the sum of the length of the message header and the length of the message payload.

6.8.56 *GetNextParam*

Function prototype:

```
uint8 *GetNextParam(uint16 paramCode)
```

Function description: This function returns the address of the next instance of the given parameter type within the current message. The parameter of this function specifies the type of message parameter.

Function parameters:

paramCode: the identification of the type of message parameter

Function returns: The function returns the pointer to the next instance of the message parameter. If it does not exist, the function returns the value *NULL*.

Note: This function cannot be used by the programmer to get the first instance of the message parameter of a given type. It assumes that the first instance has already been provided by calling the function *GetParam*. Typically, the function *GetParam* is called once to provide the first instance of the parameter and then called iteratively to provide the next instances of the parameter.

6.8.57 *GetNextParamByte*

Function prototype:

```
bool GetNextParamByte(
 uint16 paramCode,
 BYTE &param)
```

Function description: This function searches for the next instance of the given type of the single-byte parameter in the current message. If the instance is found, the function copies it into its parameter specified by the

reference and returns the value *true*; otherwise, it returns the value *false*. The parameters of this function specify the identification of the type of message parameter and the pointer to the memory area, where this function should store the next instance of the message parameter.

Function parameters:

paramCode: the identification of the type of the message parameter

param: the pointer to the memory area reserved by the programmer for the next instance of the message parameter

Function returns: This function returns the value *true* if the next instance of the message parameter is found. If the instance is not found, this function returns the value *false*.

Note: The programmer cannot use this function to get the first instance of the message parameter of the given type. This function assumes that the first instance has already been provided by calling the function *GetParamByte*. Typically, the function *GetParamByte* is called once to provide the first instance of the parameter and then called iteratively to provide the next instances of the parameter.

6.8.58 *GetNextParamDWord*

Function prototype:

```
bool GetNextParamDWord(
 uint16 paramCode,
 DWORD &param)
```

Function description: This function searches for the next instance of the given type of parameter 4 bytes in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*. The parameters of this function specify the identification of the type of the message parameter and the pointer to the memory area, where this function should store the next instance of the message parameter.

Function parameters:

paramCode: the identification of the type of message parameter

param: the pointer to the memory area reserved by the programmer for the next instance of the message parameter

Function returns: This function returns the value *true* if the next instance of the message parameter is found. If the instance is not found, this function returns the value *false*.

Note: The programmer cannot use this function to get the first instance of the message parameter of the given type. This function assumes that the first instance has already been provided by calling the function *GetParamDWord*. Typically, the function *GetParamDWord* is called once to provide the first instance of the parameter and then called iteratively to provide the next instances of the parameter.

6.8.59 *GetNextParamWord*

Function prototype:

```
bool GetNextParamWord(
 uint16 paramCode,
 WORD &param)
```

Function description: This function searches for the next instance of the given type of parameter 2 bytes in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*. The parameters of this function specify the identification of the type of the message parameter and the pointer to the memory area, where this function should store the next instance of the message parameter.

Function parameters:

paramCode: the identification of the type of message parameter

param: the pointer to the memory area reserved by the programmer for the next instance of the message parameter

Function returns: This function returns the value *true* if the next instance of the message parameter is found. If the instance is not found, this function returns the value *false*.

Note: The programmer cannot use this function to get the first instance of the message parameter of the given type. This function assumes that the first instance has already been provided by the call to the function *GetParamWord*. Typically, the function *GetParamWord* is called once to provide the first instance of the parameter and then called iteratively to provide the next instances of the parameter.

6.8.60 *GetObjectId*

Function prototype:

```
virtual uint32 GetObjectId()
```

Function description: This function returns the unique identification of this automata instance.

Function returns: This function returns the value that uniquely identifies this particular automata instance.

Note: This value has been automatically assigned to this automata instance by the function *Add*, which is called to add this automata instance to the FSM system.

6.8.61 *GetParam*

Function prototype:

```
uint8 *GetParam(uint16 paramCode)
```

Function description: This function returns the address of the first instance of the given type of message parameter within the current message. The parameter of this function specifies the identification of the parameter type.

Function parameters:

paramCode: the identification of the parameter type

Function returns: This function returns the pointer to the first instance of the message parameter within the current message. If no message parameters of the given type are found, this function returns the value *NULL*.

Note: This function returns the pointer to the beginning of the message parameter. The format of the message parameter is governed by the selected type of message information coding. For example, the parameter of the message *StandardMessage* consists of three fields. These fields are the parameter type (stored in 2 bytes), the parameter length (stored in 1 byte), and the information part of the parameter (stored in the number of bytes determined by the content of the previous field of the parameter).

6.8.62 *GetParamByte*

Function prototype:

```
bool GetParamByte(
 uint16 paramCode,
 BYTE &param)
```

Function description: This function searches for the first instance of the given type of single-byte parameter in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*. The parameters of this function specify the identification of the type of message parameter and the pointer to the memory area, where this function should store the first instance of the message parameter.

Function parameters:

paramCode: the identification of the type of message parameter

param: the pointer to the memory area reserved by the programmer for the next instance of the message parameter

Function returns: This function returns the value *true* if the first instance of the message parameter is found. If the instance is not found, this function returns the value *false*.

Note: The programmer must use this function to get the first instance of the message parameter of the given type. Typically, this function is called once to provide the first instance of the parameter, and then the function *GetNextParamByte* is called iteratively to provide the next instances of the parameter.

6.8.63 *GetParamDWord*

Function prototype:

```
bool GetParamDWord(
 uint16 paramCode,
 DWORD &param)
```

Function description: This function searches for the first instance of the given type of parameter 4 bytes in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*. The parameters of this function specify the identification of the type of message parameter and the pointer to the memory area, where this function should store the first instance of the message parameter.

Function parameters:

paramCode: the identification of the type of message parameter

param: the pointer to the memory area reserved by the programmer for the next instance of the message parameter

Function returns: This function returns the value *true* if the first instance of the message parameter is found. If the instance is not found, this function returns the value *false*.

Note: The programmer must use this function to get the first instance of the message parameter of the given type. Typically, this function is called once to provide the first instance of the parameter, and then the function *GetNextParamDWord* is called iteratively to provide the next instances of the parameter.

6.8.64 *GetParamWord*

Function prototype:

```
bool GetParamWord(
 uint16 paramCode,
 BYTE &param)
```

Function description: This function searches for the first instance of the given type of parameter 2 bytes in the current message. If the instance is found, the function copies it into its parameter specified by the reference and returns the value *true*; otherwise, it returns the value *false*. The parameters of this function specify the identification of the type of message parameter and the pointer to the memory area, where this function should store the first instance of the message parameter.

Function parameters:

paramCode: the identification of the type of message parameter

param: the pointer to the memory area reserved by the programmer for the next instance of the message parameter

Function returns: This function returns the value *true* if the first instance of the message parameter is found. If the instance is not found, this function returns the value *false*.

Note: The programmer must use this function to get the first instance of the message parameter of the given type. Typically, this function is called once to provide the first instance of the parameter, and then the function *GetNextParamWord* is called iteratively to provide the next instances of the parameter.

6.8.65 *GetProcedure*

Function prototype:

```
PROC_FUN_PTR GetProcedure(uint16 event)
```

Function description: This function returns the pointer to the event handler for the given event identifier and the current state of automata. The parameter of this function specifies the identification of the event type.

Function parameters:

event: the identification of the event type (message code)

Function returns: This function returns the pointer to the event handler. Essentially, the event handler is a C++ class function member that handles the given event type in the current state.

Note: The FSM system internal data structures contain all the necessary information about the automata states, the sets of recognizable events (messages) for all automata states, and the corresponding event handlers. This information must be defined for each automata type after it has been added to the FSM system by the function *Add*. The programmer specifies this information in the parameters of the function *Initialize*. If the event handler has not been specified by the function *Initialize* for the given event type in the current automata state, this function returns the pointer to the function *DoNothing*, which performs the default processing of the unexpected events (messages).

6.8.66 *GetRightMbx*

Function prototype:

```
virtual inline uint8 GetRightMbx()
```

Function description: This function returns the identification of the default mailbox assigned to the automata instance that is logically to the right of this automata instance.

Function returns: This function returns the number that uniquely identifies the default mailbox for the right automata instance.

Note: Historically, the terms *left* and *right* automata instance originate from SDL, where an automata instance typically communicates with its left and right neighbors. These neighbors have their own mailboxes, sometimes briefly called left and right mailboxes.

6.8.67 *GetRightAutomata*

Function prototype:

```
virtual inline uint8 GetRightAutomata()
```

Function description: This function returns the identification of the automata type that is logically to the right of this automata instance.

Function returns: This function returns the number that uniquely identifies the right automata type.

Note: By definition, right automata are logically placed to the right of the currently observed automata instance.

6.8.68 *GetRightGroup*

Function prototype:

```
virtual linline uint8 GetRightGroup()
```

Function description: This function returns the identification of the group of automata that is logically to the right of this automata instance.

Function returns: This function returns the number that uniquely identifies the right group of automata.

Note: By definition, a right group of automata is a group that contains right automata.

6.8.69 *GetRightObjectId*

Function prototype:

```
virtual inline uint32 GetRightObjectId()
```

Function description: This function returns the identification of the automata instance that is logically to the right of this automata instance.

Function returns: This function returns the number that uniquely identifies the right automata instance.

Note: By definition, right automata are logically placed to the right of the currently observed automata instance. This function returns the identification of the particular right automata instance with which the currently observed automata instance communicates.

6.8.70 *GetState*

Function prototype:

```
virtual inline uint8 GetState()
```

Function description: This function returns the identification of the current state of this automata instance.

Function returns: This function returns the value that uniquely identifies the current state of this automata instance.

6.8.71 *IsBufferSmall*

Function prototype:

```
virtual bool IsBuferSmall(
 uint8 *buff,
 uint32 length)
```

Function description: This function returns the value *true* if the size of the given buffer is not greater than the given size specified as the value of its second parameter; otherwise, it returns the value *false*. The parameters of this function specify the buffer whose size is to be checked and the size to be used as a measuring unit.

Function parameters:

buff: the pointer to the buffer whose size is to be checked

length: the value of the measuring unit

Function returns: This function returns the value *true* if the size of the given buffer is less than or equal to the given size. If the buffer size is greater than the given size, the function returns the value *false*.

6.8.72 *Initialize*

Function prototype:

```
virtual void Initialize() = 0
```

Function description: This function defines the automata state transition event handlers and timers used by this automata type. State transition event handlers are essentially the C++ functions defined by the programmer, which process events (messages). Timers are primitive time mechanisms used to restrict the duration of certain communication phases.

Note: While writing the function *Initialize*, the programmer normally defines the functions that process the expected events (messages) by calling the function *InitEventProc*, the functions that process the unexpected events by calling the function *InitUnexpectedEventProc*, and the timers by calling the function *InitTimerBlock*.

6.8.73 *InitEventProc*

Function prototype:

```
void InitEventProc(
 uint8 state,
 uint16 event,
 PROC_FUN_PTR fun)
```

Function description: This function defines the given state transition event handler for the given automata state and the given event (message code). The parameters of this function specify the identification of the state of this automata type, the identification of the event type, and the pointer to the event handler.

Function parameters:

state: the identification of the state of this automata type

event: the identification of the event type

fun: the pointer to the event handler

Note: This function may be used only within the definition of the function *Initialize*. A sequence of calls to this function fills in the internal state table for this automata type. This table is used by the FSM system and this automata type during its normal operation to locate the event handler that corresponds to the given pair (state, event).

6.8.74 *InitTimerBlock*

Function prototype:

```
void InitTimerBlock (
 uint16 tmrId,
 uint32 count,
 uint16 signalId)
```

Function description: This function initializes the given timer by the given duration and the timer expiration message code. The parameters of this function specify the timer identification, the timer duration, and the identification of the message to be sent to this automata type when the specified timer expires.

Function parameters:

tmrId: the timer identification

count: the timer duration (in timer ticks)

signalId: the identification of the message (signal) to be sent by the specified timer

Note: The timer identification is a value selected by the programmer. This value uniquely identifies the timer to the automata type that uses it in all the timer-related primitives, namely, *InitTimerBlock*, *ResetTimer*, *RestartTimer*, *StartTimer*, and *StopTimer*. Uniqueness of identifiers is limited to the scope of a single automata type. If the timer expires, it sends a special message (referred to as a *signal*) to the automata instance that has started that timer. The code of this message is set to the value of the parameter *SignalId*. The kernel calculates the absolute timer duration in seconds by dividing the time resolution specified for automata type with the time resolution of the FSM system and by multiplying this result with the basic timer resolution specified as the parameter of the function *InitKernel*.

6.8.75 *InitUnexpectedEventProc*

Function prototype:

```
void InitUnexpectedEventProc(
 uint8 state,
 PROC_FUN_PTR fun)
```

Function description: This function defines the given state transition event handler for unexpected events in the given automata state. The parameters of the function specify the automata state and the unexpected event handler, which is essentially a C++ function that handles unexpected events (messages).

Function parameters:

state: the value that uniquely identifies the automata state

fun: the pointer to the unexpected event handler

Note: If the unexpected event (message) handler does not exist because it has not been defined by this function, the FSM system and this automata type will use the function *DoNothing* to handle unexpected messages for all the states in which the unexpected message is not defined.

6.8.76 *IsTimerRunning*

Function prototype:

```
bool IsTimerRunning(uint16 id)
```

Function description: This function returns the value *true* if a given timer is active (running); otherwise, it returns the value *false*. The parameter of this function specifies the timer identification.

Function parameters:

id: the timer identification

Function returns: This function returns the value *true* if the timer is running. If the timer is not active, this function returns the value *false*.

Note: The timer may not be active because it has not been started at all, or it has been started but has expired in the meantime.

6.8.77 *NoFreeObjectProcedure*

Function prototype:

```
void NoFreeObjectProcedure(uint8 *msg)
```

Function description: This function defines the behavior of this automata type if the list of free automata of this type is used, and if it is empty at the moment when a free instance is requested. The parameter of this function specifies the pending event (message).

Function parameters:

msg: the pointer to the pending message

Note: This function is used if a group of automata of this type is used as a pool of resources of the same type. This function is called if the message related to this automata type appears and no available automata instances (resources) of this type are available. The programmer should write their own function to handle this situation in an application-specific way. This situation is additionally handled at the level of the FSM system by the function *NoFreeInstances*.

6.8.78 *NoFreeInstances*

Function prototype:

```
virtual void NoFreeInstances() = 0
```

Function description: This function defines the behavior of the FSM system if a list of free automata is used, and if it is empty at the moment when a free instance is requested.

Note: This function is used if a group of automata of this type is used as a pool of resources of the same type within the FSM system. This function is called if the message related to this automata type appears and no available automata instances (resources) of this type are available. The programmer should write their own function to handle this situation in an application-specific way. This situation is additionally handled at the level of this automata type by the function *NoFreeObjectProcedure*.

6.8.79 *ParseMessage*

Function prototype:

```
virtual bool ParseMessage(uint8 *msg)
```

Function description: This function checks if the given message is coded properly and, if it is, it becomes the current message (its pointer is assigned to the internal variable *CurrentMessage*). The parameter of this function specifies the message to be parsed.

Function parameters:

msg: the pointer to the message to be parsed

Function returns: This function returns the value *true* if the message syntax is correct; otherwise, it returns the value *false*.

Note: This function is called internally for each received message. Normally, this function is called after the reception of the message to check

its syntax. If the message syntax is correct, further message processing functions are called. Otherwise, the FSM system reports an error and discards the syntactically incorrect message.

6.8.80 *PrepareNewMessage(uint8*)*

Function prototype:

```
virtual void PrepareNewMessage(uint8 *msg)
```

Function description: This function defines the given buffer as the new message buffer by assigning the given pointer to the internal variable *NewMessage*. The buffer is used by this automata instance as a working area for the construction of the new message. The parameter of this function specifies the buffer.

Function parameters:

msg: the pointer to the buffer

Note: If the programmer wants to create a new message, they would normally call the function *GetBuffer* to obtain the buffer for the construction of the message. Next, the programmer would call this function to declare the buffer provided by the kernel as the buffer that will contain the new message. After this declaration, the programmer may use all the functions from the family of functions that operate on the new message to construct the new message. Basically, these are the *AddParamX* functions.

6.8.81 *PrepareNewMessage(uint32, uint16, uint8)*

Function prototype:

```
virtual void PrepareNewMessage(
 uint32 length,
 uint16 code,
 uint8 infoCode = LOCAL_PARAM_CODING)
```

Function description: This function creates the new message of the given length with the given message code and the given type of information coding. The parameters of this function specify the message length, the message code, and the identification of the type of message information coding.

Function parameters:

length: the message length

code: the value of the message code

infoCode: the identification of the type of message information coding

Note: Dealing with static messages of fixed and known sizes is easy. In this case, the programmer normally knows the size of the message they must create. The programmer creates the new message by calling this function and specifying the size as the value of the function parameter *length*. However, dealing with dynamic messages is more complicated, because the message length might not be known in advance. In this case, the programmer may specify the value 0 as the value of the parameter *length*. This function, in turn, will create the empty message that has its header, but has no payload. Further on, the programmer typically uses functions *AddParamX* to dynamically add new parameters to the message. Whenever not enough room exists for the new parameter in the existing new message buffer, the function *AddParamX* transparently allocates a bigger buffer, moves the content of the new message into it, and releases the smaller buffer. Of course, the price paid for this flexibility is the processing overhead for transparent buffer management.

6.8.82 *Process*

Function prototype:

```
virtual void Process(uint8 *msg)
```

Function description: This function performs the preparations for the message processing and selects the state transition event handler based on the message code and current state of this automata instance. After completion of the message processing, this function releases the buffer used by the message. The parameter of this function specifies the message to be processed.

Function parameters:

msg: the pointer to the message to be processed

Note: This function is called internally by this automata type. Because this function is virtual, the programmer may define the message handling procedure in accordance with the application-specific requirements.

6.8.83 *PurgeMailBox*

Function prototype:

```
void PurgeMailBox()
```

Function description: This function purges all the messages from the mailbox assigned to this automata type and releases all the buffers assigned to the messages.

Note: Notice that the mailbox is assigned to an automata type rather than to an individual instance of this type. This means that the mailbox may contain the messages addressed to different instances of this type. This function does not differentiate the messages. Instead, it simply purges all of them.

6.8.84 *RemoveParam*

Function prototype:

```
bool RemoveParam(uint16 paramCode)
```

Function description: This function removes the given type of message parameter from the new message. The parameter of this function specifies the identification of the type of message parameter.

Function parameters:

paramCode: the value that uniquely identifies the type of message parameter

Function returns: This function returns the value *true* if the given type of message parameter is successfully found and removed. If the new message does not contain the given type, this function returns the value *false*.

Note: Removing the type of message parameter with identification 0 is not recommended because it marks the end of the parameters in the message. The *FSMSystem* library debug version will report an error in that case and stop the program execution.

6.8.85 *Reset*

Function prototype:

```
virtual void Reset()
```

Function description: This function resets this automata instance by returning it to its initial state and stopping all its active timers.

Note: If the programmer wants to specify some additional actions to be undertaken during the restart operation, they may redefine this default behavior by writing the corresponding function member of a class derived from the class *FiniteStateMachine*.

6.8.86 *ResetTimer*

Function prototype:

```
void ResetTimer(uint16 id)
```

Function description: This function resets the internal timer block object and returns the buffer allocated by the *StartTimer* primitive to the FSM Library kernel. The parameter of this function specifies the identification of the timer.

Function parameters:

id: the value that uniquely identifies the timer

6.8.87 *RestartTimer*

Function prototype:

```
void RestartTimer(uint16 tmrId)
```

Function description: This function restarts the given timer. It is logically equivalent to a sequence of *StopTimer* and *StartTimer* primitives. The parameter of this function specifies the identification of the timer.

Function parameters:

tmrId: the value that uniquely identifies the timer

6.8.88 *RetBuffer*

Function prototype:

```
virtual void RetBuffer(uint8 *buff)
```

Function description: This function returns the given buffer to the FSM Library kernel. Normally, each memory buffer is returned at the end of its life cycle. Failure to do so leads to a memory leak problem. The parameter of this function specifies the buffer to be released.

Function parameters:

buff: the pointer to the buffer to be released

Note: The programmer must pay special attention to releasing the buffers when they are not needed anymore because the *FSMSystem* library does not include a garbage collector. Memory outage causes the exception that will stop the program execution.

6.8.89 *ReturnMsg*

Function prototype:

```
void ReturnMsg(uint8 mbxId)
```

Function description: This function makes a copy of the current message and sends it to the given mailbox. This primitive is used frequently for message forwarding. On many occasions, the communication process must react in this simple way. The parameter of this function specifies the identification of the mailbox.

Function parameters:

mbxId: the value that uniquely identifies the mailbox

6.8.90 *SetBitParamByteBasic*

Function prototype:

```
void SetBitParamByteBasic(
 BYTE param,
 uint32 offset,
 uint32 mask = MASK_32_BIT)
```

Function description: This function sets the given single-byte parameter of the new message to the result of the bit-wise inclusive OR operation applied to the given parameter and its previous value masked (bit-wise AND operation) with the given bit-mask. The parameters of this function specify the value of the single-byte parameter, the offset of the target parameter of the new message, and the value of the bit-mask.

Function parameters:

param: the value of the single-byte parameter

offset: the target parameter of the new message

mask: the value of the bit-mask

6.8.91 *SetBitParamDWordBasic*

Function prototype:

```
void SetBitParamDWordBasic(
 DWORD param,
 uint32 offset,
 uint32 mask = MASK_32_BIT)
```

Function description: This function sets the given 4-byte parameter of the new message to the result of the bit-wise inclusive OR operation applied to the given parameter and its previous value masked (bit-wise AND operation) with the given bit-mask. The parameters of this function specify the value of the 4-byte parameter, the offset of the target parameter of the new message, and the value of the bit-mask.

Function parameters:

param: the value of the 4-byte parameter

offset: the target parameter of the new message

mask: the value of the bit-mask

6.8.92 SetBitParamWordBasic

Function prototype:

```
void SetBitParamWordBasic(
 WORD param,
 uint32 offset,
 uint32 mask = MASK_32_BIT)
```

Function description: This function sets the given 2-byte parameter of the new message to the result of the bit-wise inclusive OR operation applied to the given parameter and its previous value masked (bit-wise AND operation) with the given bit-mask. The parameters of this function specify the value of the 2-byte parameter, the offset of the target parameter of the new message, and the value of the bit-mask.

Function parameters:

param: the value of the 2-byte parameter

offset: the target parameter of the new message

mask: the value of the bit-mask

6.8.93 *SetCallId()*

Function prototype:

```
inline void SetCallId()
```

Function description: This function sets the default value of the attribute *CallId* of this automata instance.

Note: This function automatically allocates the first available identification and assigns it to the protected class attribute *CallId*, completely transparent to the programmer.

6.8.94 *SetCallId(uint32)*

Function prototype:

```
inline void SetCallId(uint32 id)
```

Function description: This function sets the given value of the attribute *CallId* of this automata instance. The parameter of this function specifies the value to be assigned to the attribute *CallId*.

Function parameters:

id: the value to be assigned to the attribute *CallId*

Note: In contrast to an overloaded function without any parameters in its signature, this function enables the programmer to manually assign the value to the attribute *CallId*. However, this value must be unique. The programmer must pay special attention to the assignment of these numbers, especially if they mix this function call with function calls to the overloaded function that assigns the default values.

6.8.95 *SetCallIdFromMsg*

Function prototype:

```
inline void SetCallIdFromMsg()
```

Function description: This function sets the attribute *CallId* of this automata instance to the value of the parameter *CallId* of the current message. This primitive is used to store the reference number specific to the communication protocol.

6.8.96 *SetDefaultFSMData*

Function prototype:

```
virtual void SetDefaultFSMData() = 0
```

Function description: This function sets the automata-specific data to their default values. It is typically used before the normal operation phase.

Note: The programmer must define this virtual function for a class derived from the class *FiniteStateMachine*. They do so by writing a C++ function that initializes the problem-specific data.

6.8.97 *SetDefaultHeader*

Function prototype:

```
virtual void SetDefaultHeader(uint8 infoCoding = 0)
```

Function description: This function sets the default header field values for the given type of message information coding. The parameter of this function specifies the identification of the type of message information coding.

Function parameters:

infoCoding: the type of message information coding

Note: The programmer must define this virtual function for a class derived from the class *FiniteStateMachine*. They do so by writing a C++ function that fills in the protocol-specific data in the new message header.

6.8.98 *SetGroup*

Function prototype:

```
inline void SetGroup(uint8 id)
```

Function description: This function sets the identification of the group of automata for this automata type to the given value. This primitive is used to declare group membership. The parameter of this function specifies the value to be assigned to the corresponding class attribute.

Function parameters:

id: the value that uniquely identifies the group of automata

6.8.99 *SetInitialState*

Function prototype:

```
virtual void SetInitialState()
```

Function description: This function sets the current state of this automata instance to its initial state.

Note: The programmer must obey the rule that the value of the identification of the initial automata state is 0.

6.8.100 *SetKernelObjects*

Function prototype:

```
static void SetKernelObjects(
 TPostOffice *postOffice,
 TBuffers *buffers,
 CTimer *timer)
```

Function description: This function sets the *FSMSystem* library kernel objects (post office, buffers, and timers), which are common for all the automata in the FSM system. The parameters of this function specify the post office object, the buffers object, and the timers object.

Function parameters:

postOffice: the pointer to the post office object

buffers: the pointer to the buffers object

timer: the pointer to the timers object

Note: This function is called internally by the function *InitKernel*. Remember that this function defines the kernel objects that are common for all automata types and all their instances. An accidental call to this function may cause unpredictable behavior in the FSM system.

6.8.101 *SetLeftMbx*

Function prototype:

```
inline void SetLeftMbx(uint8 mbx)
```

Function description: This function sets the default identification of the mailbox assigned to the automata instance that is logically to the left of this automata instance. The parameter of this function specifies the identification of the mailbox.

Function parameters:

mbx: the value that uniquely identifies the mailbox

6.8.102 *SetLeftAutomata*

Function prototype:

```
inline void SetLeftAutomata(uint8 automata)
```

Function description: This function sets the identification of the automata type that is logically to the left of this automata instance. The parameter of this function specifies the identification of the automata type.

Function parameters:

automata: the value that uniquely identifies the automata type

6.8.103 *SetLeftObject*

Function prototype:

```
inline void SetLeftObject(uint8 group)
```

Function description: This function sets the identification of the type of the group of automata that is logically to the left of this automata instance. The parameter of this function specifies the identification of the group of automata.

Function parameters:

group: the value that uniquely identifies the group of automata

6.8.104 *SetLeftObjectId*

Function prototype:

```
inline void SetLeftObjectId(uint32 id)
```

Function description: This function sets the identification of the automata instance that is logically to the left of this automata instance. The parameter of this function specifies the identification of the automata instance.

Function parameters:

id: the identification of the automata instance

6.8.105 *SetLogInterface*

Function prototype:

```
static void SetLogInterface(LogInterface *logingObject)
```

Function description: This function defines the object responsible for message logging. The object is an instance of a class derived from the class *LogInterface*. The parameter of this function specifies the message logging object.

Function parameters:

logingObject: the pointer to the message logging object

Note: The programmer must not call this function before the initialization of all the automata included in the FSM system has been finished. The logging object may log data to the file on the local mass memory unit (e.g., flash memory) or to the network file server. The log file is essential for debugging and test and verification purposes.

6.8.106 *SendMessage(uint8)*

Function prototype:

```
inline void SendMessage(uint8 mbxId)
```

Function description: This function sends the new message to the given mailbox. The parameter of this function specifies the identification of the mailbox.

Function parameters:

mbxId: the value that uniquely specifies the mailbox

Note: By definition, the internal pointer *NewMessage* points to the buffer that contains the new message. The programmer initializes this pointer by calling the function *PrepareNewMessage*.

6.8.107 *SendMessage(uint8, uint8*)*

Function prototype:

```
inline void SendMessage(
 uint8 mbxId,
 uint8 *msg)
```

Function description: This function sends the given message to the given mailbox. The parameters of this function specify the identification of the mailbox and the message to be sent to that mailbox.

Function parameters:

mbxId: the value that uniquely identifies the mailbox

msg: the pointer to the message

6.8.108 *SetMessageFromData*

Function prototype:

```
void SetMessageFromData()
```

Function description: This function sets the header fields of the new message related to the originating automata instance to the values specific to this automata instance. The data specifying the originating automata instance are its type, group, and identification.

Note: This function is automatically called from the function *SendMessage*.

6.8.109 *SetMsgCallId(uint32)*

Function prototype:

```
inline void SetMsgCallId(uint32 id)
```

Function description: This function sets the call ID parameter of the new message to the given value. The parameter of this function specifies the value of the call ID.

Function parameters:

id: the value of the call ID

Note: The call ID parameter has been traditionally used to identify a single telephone call. In general, it may be used to uniquely identify a communication process or a transaction that engages a group of automata that participates in its processing.

6.8.110 *SetMsgCallId(unit32, unit8*)*

Function prototype:

```
inline void SetMsgCallId(
 uint32 id,
 uint8 *msg)
```

Function description: This function sets the call ID parameter of the given message to the given value. The parameters of this function specify the value of the call ID and the target message.

Function parameters:

id: the value of the call ID

msg: the pointer to the buffer that contains the target message

Note: The value of the call ID parameter is the same for all the messages involved in a transaction or a process, e.g., a single telephone call.

6.8.111 *SetMsgCode(uint16)*

Function prototype:

```
inline void SetMsgCode(uint16 code)
```

Function description: This function sets the message code parameter of the new message to the given value. The parameter of this message specifies the message code.

Function parameters:

code: the message code

6.8.112 *SetMsgCode(uint16, uint8*)*

Function prototype:

```
inline void SetMsgCode(
 uint16 code,
 uint8 *msg)
```

Function description: This function sets the message code parameter of the given message to the given value. The parameters of this function specify the message code and the target message.

Function parameters:

code: the message code

msg: the pointer to the buffer that contains the target message

6.8.113 *SetMsgFromAutomata(uint8)*

Function prototype:

```
inline void SetMsgFromAutomata(uint8 from)
```

Function description: This function sets the type of the originating automata parameter of the new message to the given value. The parameter of this function specifies the identification of the automata type that is the message source.

Function parameters:

from: the identification of the automata type

Note: This function is automatically called by the function *SetMessage FromData*.

6.8.114 *SetMsgFromAutomata(uint8, uint8*)*

Function prototype:

```
inline void SetMsgFromAutomata(
 uint8 from,
 uint8 *msg)
```

Function description: This function sets the type of the originating automata parameter of the given message to the given value. The parameters of this function specify the type of automata that is the message source and the target message.

Function parameters:

from: the automata type that is the message source

msg: the pointer to the buffer that contains the target message

6.8.115 *SetMsgFromGroup(uint8)*

Function prototype:

```
inline void SetMsgFromGroup(uint8 from)
```

Function description: This function sets the type of the originating group of automata parameter of the new message to the given value. The parameter of this message specifies the identification of the group of automata that is the message source.

Function parameters:

from: the identification of the group of automata that is the message source

Note: This function is automatically called by the function *SetMessage FromData*.

6.8.116 *SetMsgFromGroup(uint8, uint8*)*

Function prototype:

```
inline void SetMsgFromGroup(
 uint8 from,
 uint8 *msg)
```

Function description: This function sets the type of the originating group of automata parameter of the given message to the given value. The parameters of this function specify the identification of the group of automata that is the message source and the target message.

Function parameters:

from: the identification of the group of automata that is the message source

msg: the pointer to the buffer that contains the target message

6.8.117 *SetMsgInfoCoding(uint8)*

Function prototype:

```
inline void SetMsgInfoCoding(uint8 codingType)
```

Function description: This function sets the message information coding parameter of the new message to the given value. The parameter of this message specifies the identification of the information coding scheme.

Function parameters:

codingType: the value that uniquely specifies the information coding scheme

Note: This function is automatically called by the function *PrepareNewMessage*.

6.8.118 *SetMsgInfoCoding(uint8, uint8*)*

Function prototype:

```
inline void SetMsgInfoCoding(
 uint8 codingType,
 uint8 *msg)
```

Function description: This function sets the message information coding parameter of the given message to the given value. The parameters of this function specify the identification of the information coding scheme and the target message.

Function parameters:

codingType: the identification of the information coding scheme

msg: the pointer to the target message

6.8.119 *SetMsgInfoLength(uint16)*

Function prototype:

```
inline void SetMsgInfoLength(uint16 length)
```

Function description: This function sets the message payload (useful information) length parameter of the new message. The parameter of this function specifies the value of the payload length.

Function parameters:

length: the payload length in octets (bytes)

Note: All the *AddParamX* functions—which are responsible for adding parameters to the new message—call this function automatically to update the length of the message payload.

6.8.120 *SetMsgInfoLength(uint16, uint8*)*

Function prototype:

```
inline void SetMsgInfoLength(
 uint16 length,
 uint8 *msg)
```

Function description: This function sets the message payload (useful information) length parameter of the given message. The parameters of this function specify the value of the payload length and the target message.

Function parameters:

length: the payload length in octets (bytes)

msg: the pointer to the buffer that contains the target message

6.8.121 *SetMsgObjectNumberFrom(uint32)*

Function prototype:

```
inline void SetMsgObjectNumberFrom(uint32 from)
```

Function description: This function sets the originating automata instance identification parameter of the new message to the given value. The parameter of this function specifies the identification of the automata instance that is the message source.

Function parameters:

from: the identification of the automata instance that is the message source

Note: This function is automatically called by the function *SetMessage FromData*.

6.8.122 *SetMsgObjectNumberFrom(uint32, uint8*)*

Function prototype:

```
inline void SetMsgObjectNumberFrom(
 uint32 from,
 uint8 *msg)
```

Function description: This function sets the originating automata instance identification parameter of the given message to the given value. The parameters of this message specify the identification of the automata instance that is the message source and the target message.

Function parameters:

from: the identification of the automata instance that is the message source

msg: the pointer to the buffer that contains the target message

6.8.123 *SetMsgObjectNumberTo(uint32)*

Function prototype:

```
inline void SetMsgObjectNumberTo(uint32 to)
```

Function description: This function sets the destination automata instance identification parameter of the new message to the given value. The

parameter of this function specifies the automata instance that is the message destination.

Function parameters:

to: the automata instance that is the message destination

6.8.124 *SetMsgObjectNumberTo(uint32, uint8*)*

Function prototype:

```
inline void SetMsgObjectNumberTo(uint32 to,uint8 *msg)
```

Function description: This function sets the destination automata instance identification parameter of the given message to the given value. The parameters of this function specify the automata instance that is the message destination and the target message.

Function parameters:

to: the automata instance that is the message destination

msg: the pointer to the buffer that contains the target message

6.8.125 *SetMsgToAutomata(uint8)*

Function prototype:

```
inline void SetMsgToAutomata(uint8 to)
```

Function description: This function sets the destination automata type identification parameter of the new message to the given value. The parameter of this function specifies the automata type that is the message destination.

Function parameters:

to: the automata type that is the message destination

6.8.126 *SetMsgToAutomata(uint8, uint8*)*

Function prototype:

```
inline void SetMsgToAutomata(
 uint8 to,
 uint8 *msg)
```

Function description: This function sets the destination automata type identification parameter of the given message to the given value. The

parameters of this function specify the identification of the automata type that is the message destination and the target message.

Function parameters:

to: the identification of the automata type that is the message destination

msg: the pointer to the buffer that contains the target message

6.8.127 *SetMsgToGroup(uint8)*

Function prototype:

```
inline void SetMsgToGroup(uint8 to)
```

Function description: This function sets the destination automata group identification parameter of the new message to the given value. The parameter of this function specifies the identification of the group of automata that is the message destination.

Function parameters:

to: the identification of the group of automata that is the message destination

6.8.128 *SetMsgToGroup(uint8, uint8*)*

Function prototype:

```
inline void SetMsgToGroup(
 uint8 to,
 uint8 *msg)
```

Function description: This function sets the destination automata group identification parameter of the given message to the given value. The parameters of this function specify the identification of the group of automata that is the message destination and the target message.

Function parameters:

to: the identification of the group of automata that is the message destination

msg: the pointer to the buffer that contains the target message

6.8.129 *SendMessageLeft*

Function prototype:

```
void SendMessageLeft()
```

Function description: This function sends the new message to the mailbox assigned to the automata instance that is logically to the left of this automata instance.

Note: The programmer may use this function if they have already defined the left automata instance for the currently observed automata instance. This definition includes the definition of the mailbox assigned to the left automata instance. If the left automata instance and its mailbox are defined, this function automatically fills in all the data related to both the source (originating) and destination automata instances within the new message and sends the new message to the left mailbox.

6.8.130 *SendMessageRight*

Function prototype:

```
void SendMessageLeft()
```

Function description: This function sends the new message to the mailbox assigned to the automata instance that is logically to the right of this automata instance.

Note: The programmer may use this function if they have already defined the right automata instance for the currently observed automata instance. This definition includes the definition of the mailbox assigned to the right automata instance. If the right automata instance and its mailbox are defined, this function automatically fills in all the data related to both the source (originating) and destination automata instances within the new message and sends the new message to the right mailbox.

6.8.131 *SetNewMessage*

Function prototype:

```
inline void SetNewMessage(uint8 *msg)
```

Function description: This function sets the new message to the given message by assigning the given message pointer to the internal pointer to the new message. The parameter of this function specifies the target message.

Function parameters:

msg: the pointer to the buffer that contains the target message

6.8.132 *SetObjectId*

Function prototype:

```
inline void SetObjectId(uint32 id)
```

Function description: This function sets the identification of this automata instance to the given value. The parameter of this function specifies the identification of this automata instance.

Function parameters:

id: the value that uniquely identifies this automata instance

6.8.133 *SetRightMbx*

Function prototype:

```
inline void SetRightMbx(uint8 mbx)
```

Function description: This function sets the identification of the mailbox assigned to the automata instance that is logically to the right of this automata instance. The parameter of this message specifies the identification of the right mailbox for this automata instance.

Function parameters:

mbx: the identification of the right mailbox for this automata instance

6.8.134 *SetRightAutomata*

Function prototype:

```
inline void SetRightAutomata(uint8 automata)
```

Function description: This function sets the identification of the automata type that is logically to the right of this automata instance. The parameter of this function specifies the automata type that is to the right of this automata instance.

Function parameters:

automata: the identification of the automata type

6.8.135 *SetRightObject*

Function prototype:

```
inline void SetRightObject(uint8 group)
```

Function description: This function sets the identification of the type of the group of automata that is logically to the right of this automata instance. The parameter of this function specifies the type of the group of automata that is to the right of this automata instance.

Function parameters:

group: the identification of the group of automata

6.8.136 *SetRightObjectId*

Function prototype:

```
inline void SetRightObjectId(uint32 id)
```

Function description: This function sets the identification of the automata instance that is logically to the right of this automata instance. The parameter of this function specifies the identification of the automata instance that is to the right of this automata instance.

Function parameters:

id: the identification of the automata instance

6.8.137 *SetState*

Function prototype:

```
inline void SetState(uint8 state)
```

Function description: This function sets the identification of the current state of this automata instance. The parameter of this function specifies the identification of the state.

Function parameters:

state: the value that uniquely identifies the particular state of automata

6.8.138 *StartTimer*

Function prototype:

```
void StartTimer(uint16 tmrId)
```

Function description: This function starts the given timer. The parameter of this function specifies the identification of the timer.

Function parameters:

tmrId: the value that uniquely identifies the particular timer

Note: Uniqueness of the timer identifier is limited to the scope of a single automata type that uses it.

6.8.139 *StopTimer*

Function prototype:

```
void StopTimer(uint16 tmrId)
```

Function description: This function stops the given timer. The parameter of this function specifies the identification of the timer.

Function parameters:

tmrId: the value that uniquely identifies the particular timer

Note: Uniqueness of the timer identifier is limited to the scope of a single automata type that uses it.

6.8.140 *SysClearLogFlag*

Function prototype:

```
static void SysClearLogFlag()
```

Function description: This function stops the logging of the messages exchanged by the automata.

6.8.141 *SysStartAll*

Function prototype:

```
Static void SysStartAll()
```

Function description: This function starts the logging of the messages exchanged by the automata.

Note: Normally, the programmer should start the logging of messages before they start the individual automata included in the FSM system.

6.8.142 *NetFSM*

Function prototype:

```
NetFSM(
 uint16 numOfTimers = DEFAULT_TIMER_NO,
 uint16 numOfState = DEFAULT_STATE_NO,
 uint16 maxNumOfProceduresPerState = DEFAULT_PROCEDURE_NO_PER_STATE,
 bool getMemory = true)
```

Function description: This constructor initializes the object that represents an instance of the given automata type together with the data structures needed for its proper operation. The parameters of this function specify the number of timers to be used by this automata type, the total number of states for this automata type, the maximal number of state transitions per state for this automata type, and the memory allocation indicator. All the parameters have their default values as shown in the function prototype declaration above.

Function parameters:

numOfTimers: the number of timers to be used by this automata type

numOfState: the total number of states for this automata type

maxNumOfProceduresPerState: the maximal number of state transitions per state

getMemory: the memory allocation indicator

Note: The programmer may call this a constructor without parameters. In this case, the parameters will be set to their corresponding default values. The value of the fourth parameter *getMemory* regulates memory allocation. By default, this indicator is set to the value *true*, which means that the constructor will take care of memory allocation. Default memory allocation is not optimal because it is based on the maximal number of transitions per state. This compromise has been made intentionally because it leads to a very simple FSM definition API. If the programmer wants to optimize memory allocation, they may build the data structure describing the FSM by allocating necessary memory blocks from the memory heap, linking them together, and storing the pointer to this data structure in the protected class field member *States* before this function is called. In that case, the programmer would set the fourth parameter *getMemory* to the value *false*.

6.8.143 *convertFSMToNetMessage*

Function prototype:

```
virtual void convertFSMToNetMessage() = 0
```

Function description: This function converts the internal message format into the external message format appropriate for the transmission over the TCP/IP network.

Note: The programmer must define this virtual function by writing the corresponding function member of a class derived from the class *NetFSM*.

6.8.144 *convertNetToFSMMessage*

Function prototype:

```
virtual uint16 convertNetToFSMMessage() = 0
```

Function description: This function converts the external message format into the internal message format appropriate for the communication within the FSM system.

Function returns: This function returns the code of the received message.

Note: The programmer must define this virtual function by writing the corresponding function member of a class derived from the class *NetFSM*.

6.8.145 *establishConnection*

Function prototype:

```
void establishConnection()
```

Function description: This function establishes the TCP connection between two geographically distributed FSM systems.

Note: The programmer must call this function before they can call the function *sendToTCP* to send the message to the remote FSM system.

6.8.146 *getProtocolInfoCoding*

Function prototype:

```
virtual uint8 getProtocolInfoCoding() = 0
```

Function description: This function returns the identification of the type of external message coding.

Function returns: This function returns the value that uniquely identifies the type of coding of the external message.

6.8.147 *sendToTCP*

Function prototype:

```
void sendToTCP()
```

Function description: This function sends the new message to the remote FSM system over the previously established TCP connection.

Note: The programmer must call the function *establishConnection* before they can call this function.

6.9 A Simple Example with Three Automata Instances

This section shows how the programmer can construct the FSM system and how they can add individual automata instances to it. To keep the example simple, we include only one use case, *Show Simple Demo* (Figure 6.1). The realization of this use case is a simple collaboration that comprises three instances (*instance_1*, *instance_2*, and *instance_3*) of the same automata type

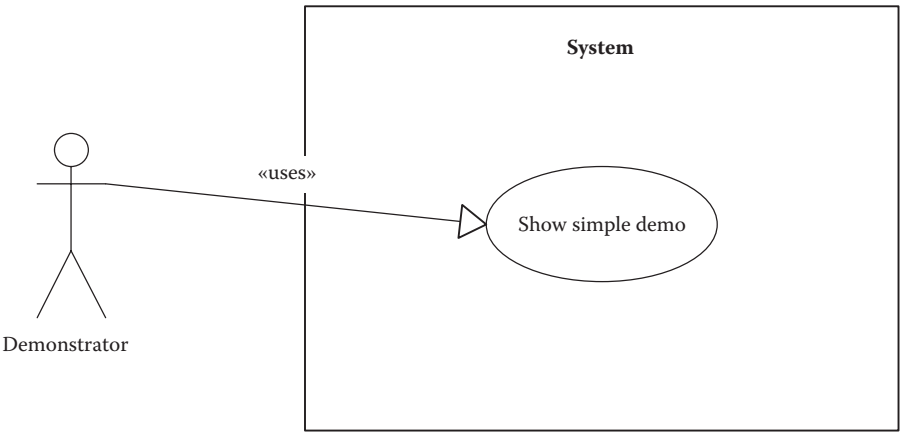


FIGURE 6.1
Simple use case diagram for the example with three automata instances.

(*Automata*), which are added to the FSM system (Figure 6.2). These three automata instances have the trivial task of exchanging the given number of messages in a "round robin" fashion.

At the beginning, the main thread calls the function *StartDemo* of *instance_1*, which, in turn, asynchronously sends itself the message *IDLE_START*. Upon reception of this message, *instance_1* sends the message *IDLE_MSG* to *instance_2*, which increments the message sequence number and forwards the message to *instance_3*; the latter translates it to the message *MSG_MSG* and sends it back to *instance_1*. This message then makes two full circles around the collaborating objects. Finally, *instance_1* translates it to the message *MSG_STOP* and sends it to *instance_2*, which, in turn, forwards it to *instance_3*. The corresponding sequence diagram is shown in Figure 6.3. The conditions A, B, and C regulate the already mentioned translations of the messages.

The statechart diagram that describes the behavior of a single automata instance is organized into two hierarchical levels. The top level comprises two simple states (*IDLE* and *MESSAGE*) and four composite states (*Automata_IDLE_START*, *Automata_IDLE_MSG*, *Automata_MSG_MSG*, and *Automata_MSG_STOP*) (Figure 6.4). The symbolic constant *MAX_MSG_NUM* is defined to have the value 10 in this example. The variable *msgno* is the message sequence number, whose values are shown in parentheses in Figures 6.2 and 6.3. Later in the program text, this short variable name suitable for figures is replaced with the longer self-documenting name *msgNumber*.

The individual composite states *Automata_IDLE_START*, *Automata_IDLE_MSG*, *Automata_MSG_MSG*, and *Automata_MSG_STOP* are shown in

FIGURE 6.2
Collaboration diagram for the example with three automata instances.

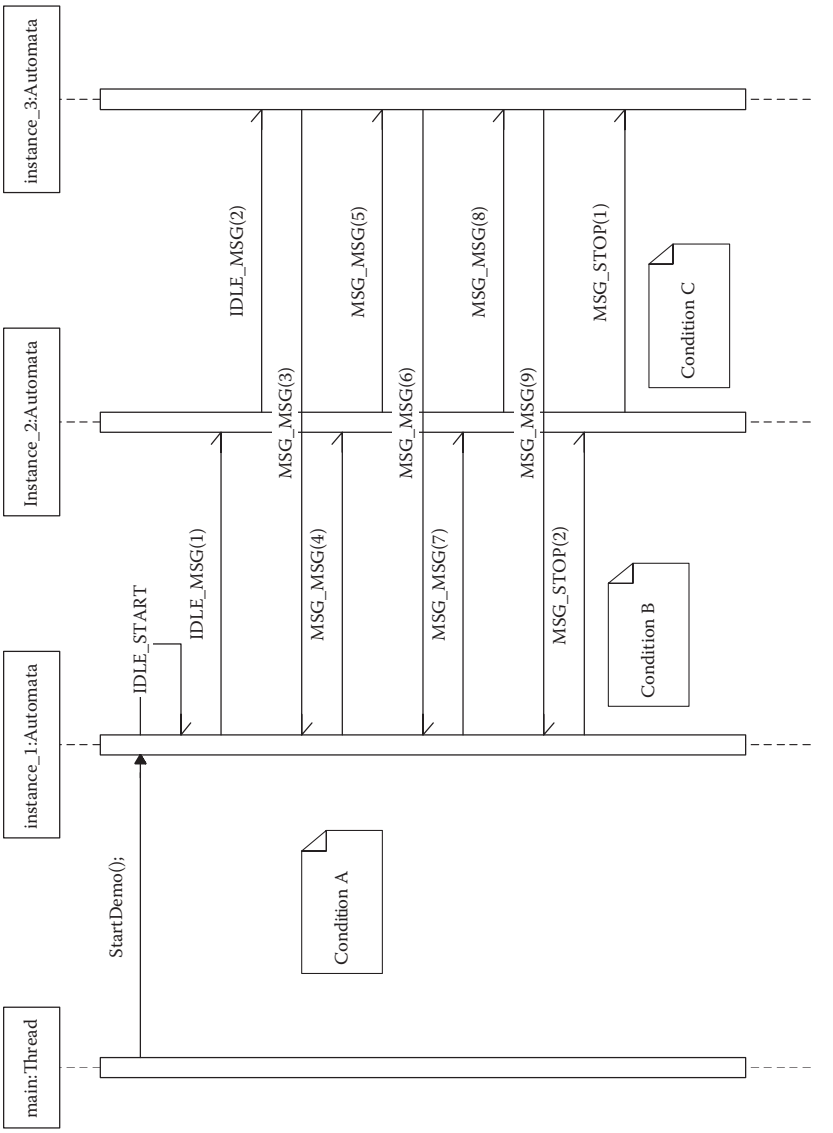


FIGURE 6.3
Sequence diagram for the example with three automata instances.

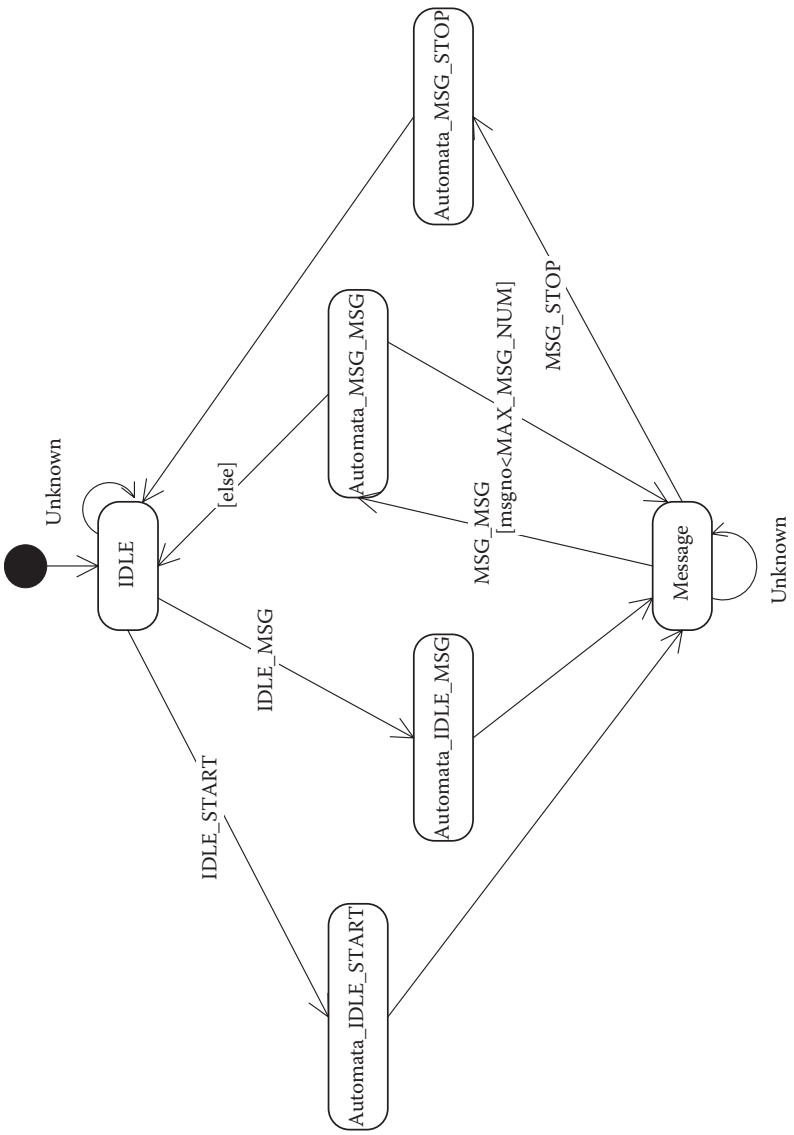


FIGURE 6.4
Statechart diagram for the example with three automata instances.

Figures 6.5 through 6.8, respectively. These have been made rather detailed to show how to provide the mapping from the UML model to the corresponding program code by the application of forward engineering. Essentially, the state transition actions are sequences of calls to functions provided by the FSM Library, such as *PrepareNewMessage*, *AddParamDWord*, *SendMessage*, and so on.

Each of the composite states can be modeled as an operation by the corresponding activity diagram. The activity diagrams for the operations *Automata_IDLE_START*, *Automata_IDLE_MSG*, *Automata_MSG_MSG*, and *Automata_MSG_STOP* are shown in Figures 6.9 through 6.12, respectively. Again, these diagrams have been made by applying forward engineering, but on a slightly higher abstraction level, using informal text statements instead of explicit functions calls. Essentially, composite statechart and activity diagrams have the same semantics in this example.

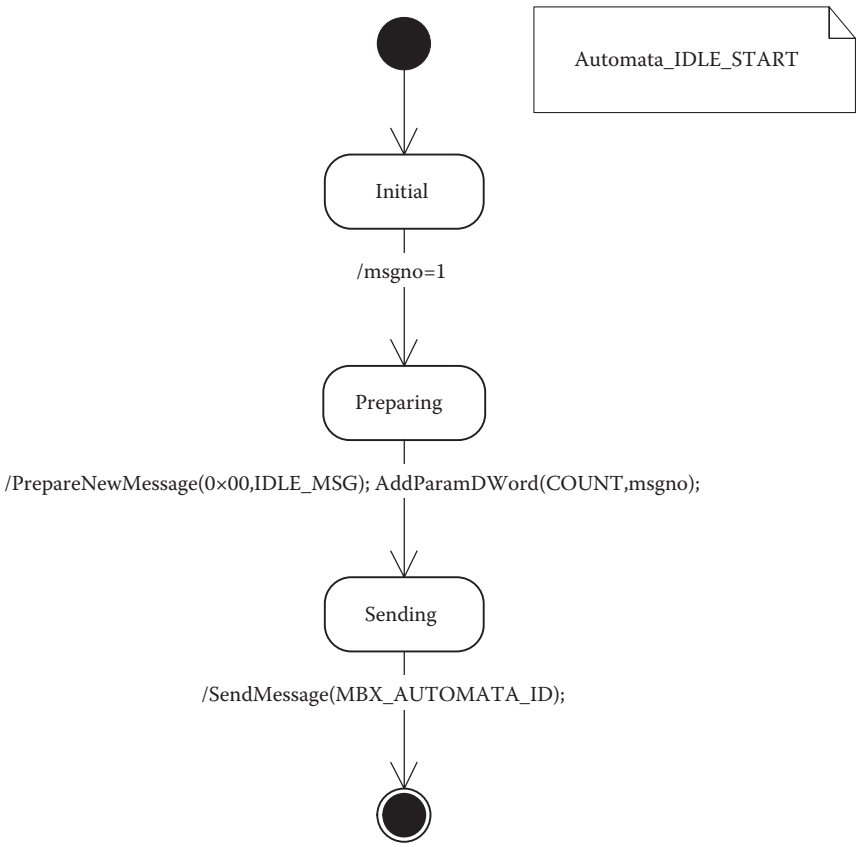


FIGURE 6.5
Statechart diagram for the composite state *Automata_IDLE_START*.

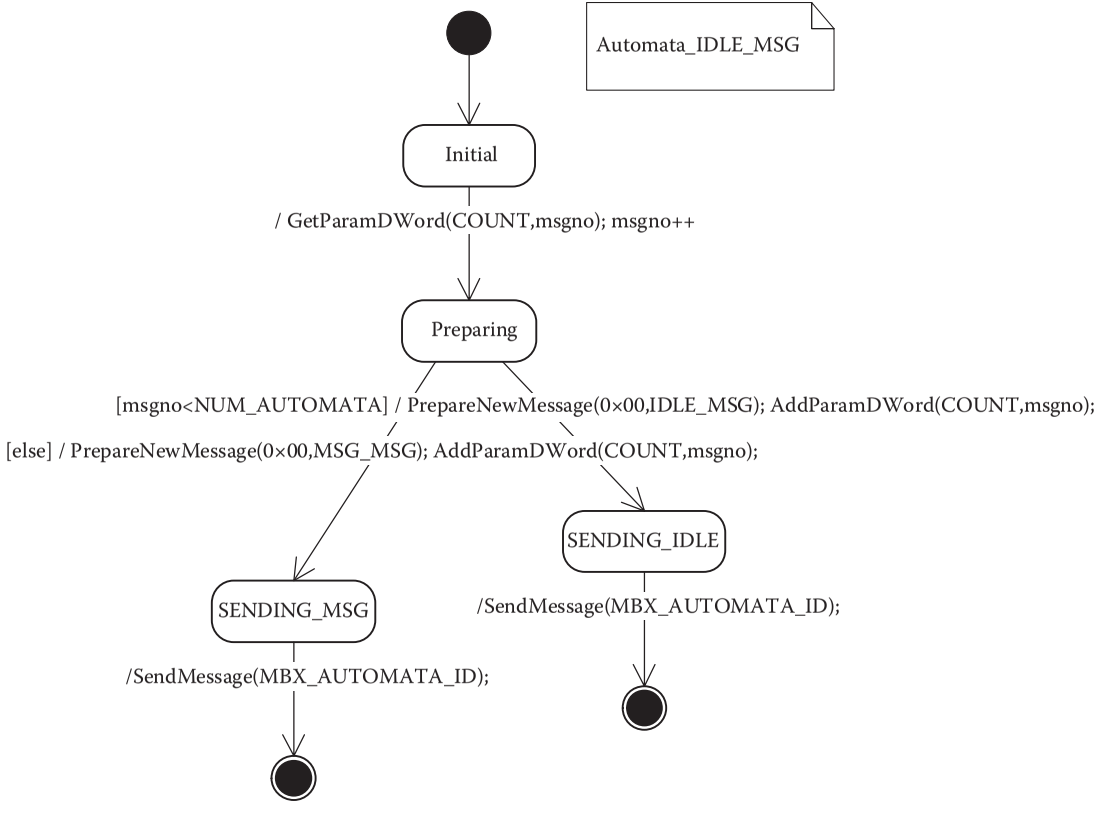


FIGURE 6.6
Statechart diagram for the composite state *Automata_IDLE_MSG*.

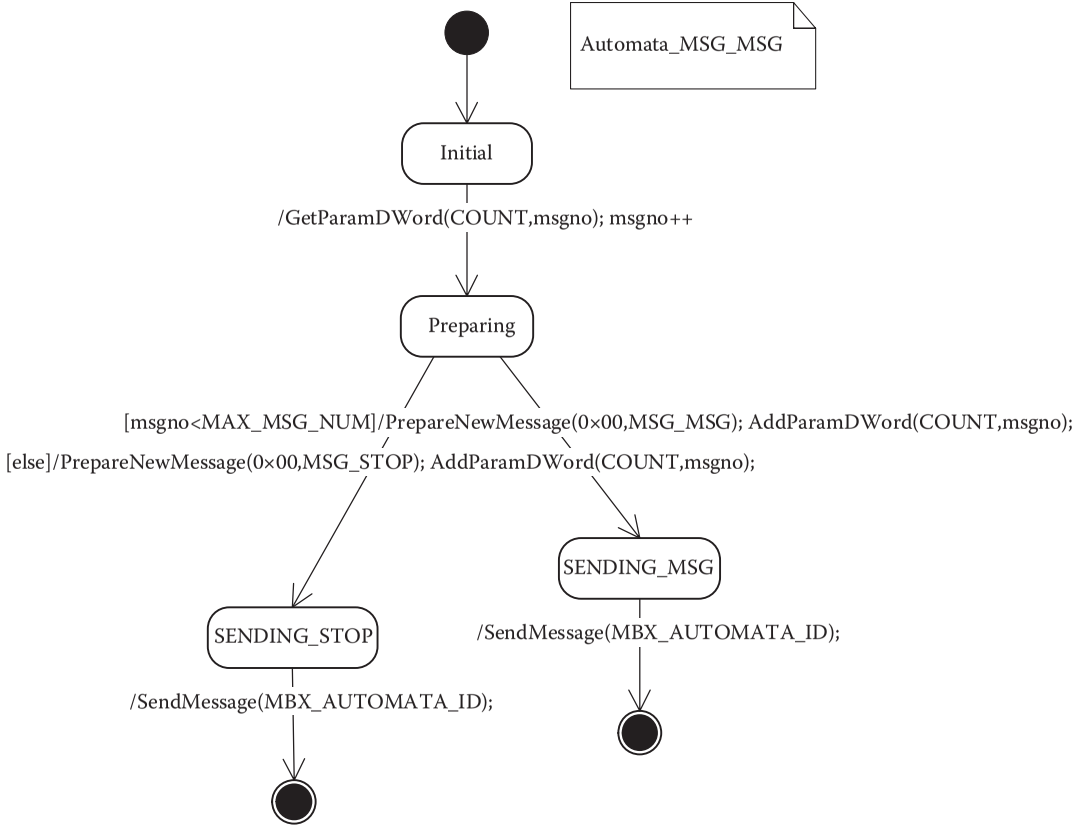


FIGURE 6.7
Statechart diagram for the composite state *Automata_MSG_MSG*.

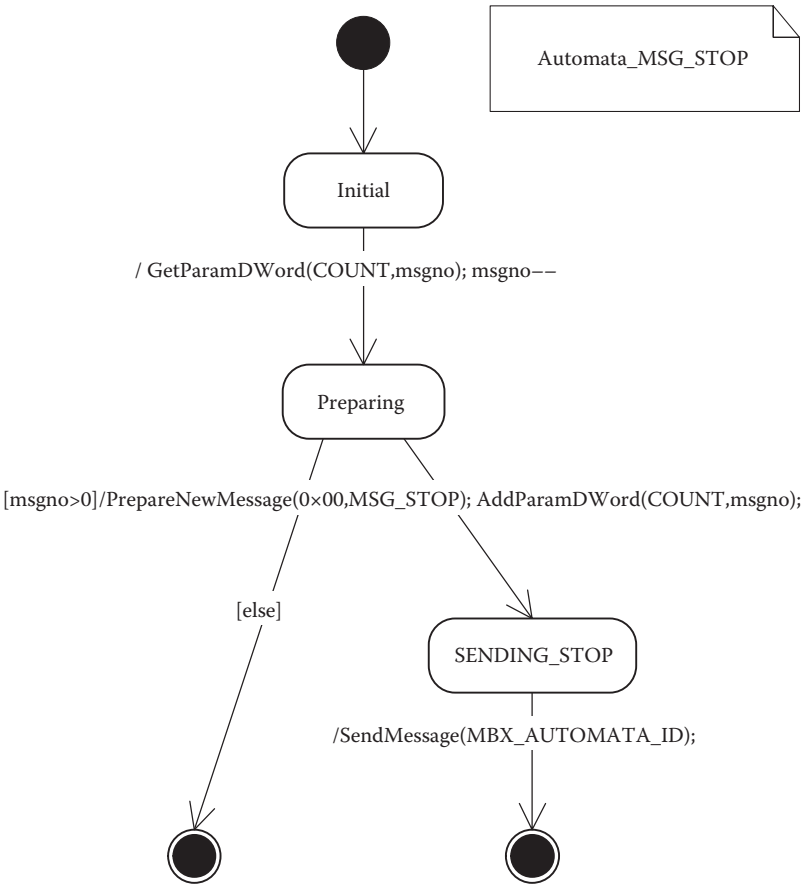


FIGURE 6.8
Statechart diagram for the composite state *Automata_MSG_STOP*.

The third and semantically equivalent method of modeling the behavior of individual automata instances is by using the domain-specific SDL model. This model comprises state transitions triggered by the reception of the corresponding messages. The same names are used again so that the reader can easily follow the correspondence between the SDL state transitions and the UML composite states and activity diagrams. The SDL state transitions *Automata_IDLE_START*, *Automata_IDLE_MSG*, *Automata_MSG_MSG*, and *Automata_MSG_STOP* are shown in Figures 6.13 through 6.16, respectively.

As already mentioned, all three automata instances in this example are of the same type, i.e., class. The class *Automata* is a specialization of the FSM Library class *FiniteStateMachine* and is used by the FSM Library class *FSMSystem* (see the corresponding UML class diagram in Figure 6.17). The

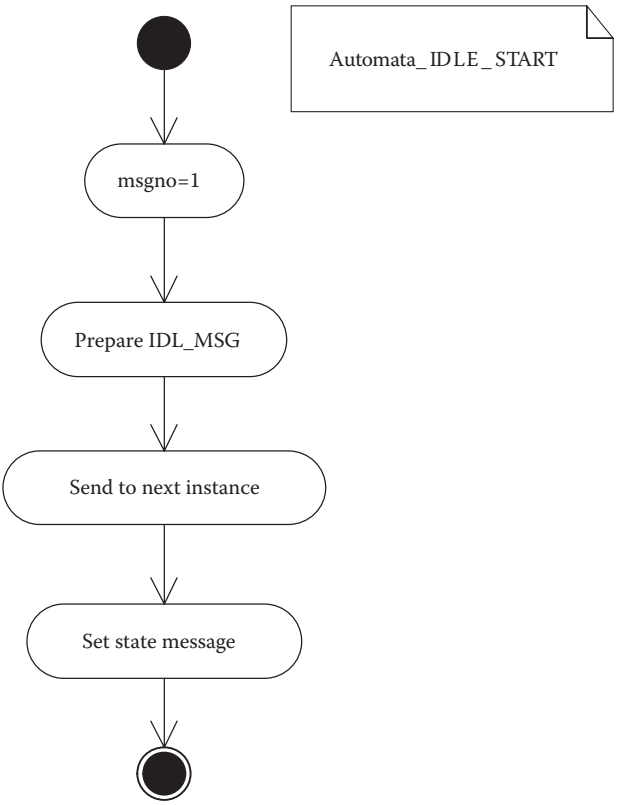


FIGURE 6.9
Activity diagram for the operation *Automata_IDLE_START*.

class *Automata* inherits all the members from its parent class and adds some field members (such as *msgno*) and function members (such as *Automata_IDLE_START*, *Automata_IDLE_MSG*, *Automata_MSG_MSG*, *Automata_MSG_STOP*, *Initialize*, and *StartDemo*). The first four correspond to composite states from the previous UML statechart model.

An object diagram, such as the one shown in Figure 6.18, helps us to better understand the structural relationships among objects. A collaboration diagram (Figure 6.2) shows the logical communication of automata instances over their virtual, peer-to-peer connections. On a more detailed level of abstraction, we see that the real communication is governed by the FSM system, which is the owner of the mailboxes (not shown in the figure) used for storing the messages, e.g., *StandardMessage* (shown in Figure 6.18). This particular message shown in one snapshot of object collaboration is the first message sent from *instance_1* to *instance_2*. The message code is *IDLE_MSG*, and the value of the message sequence parameter is 1.

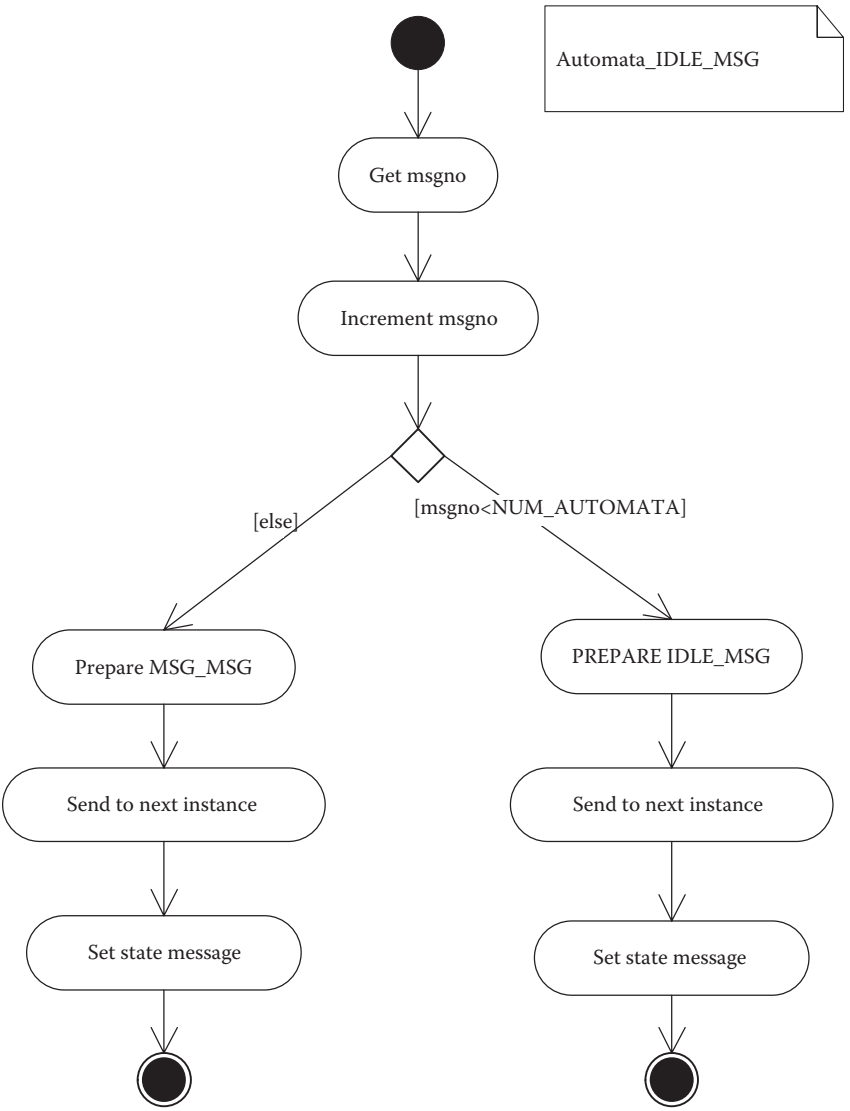


FIGURE 6.10
Activity diagram for the operation *Automata_IDLE_MSG*.

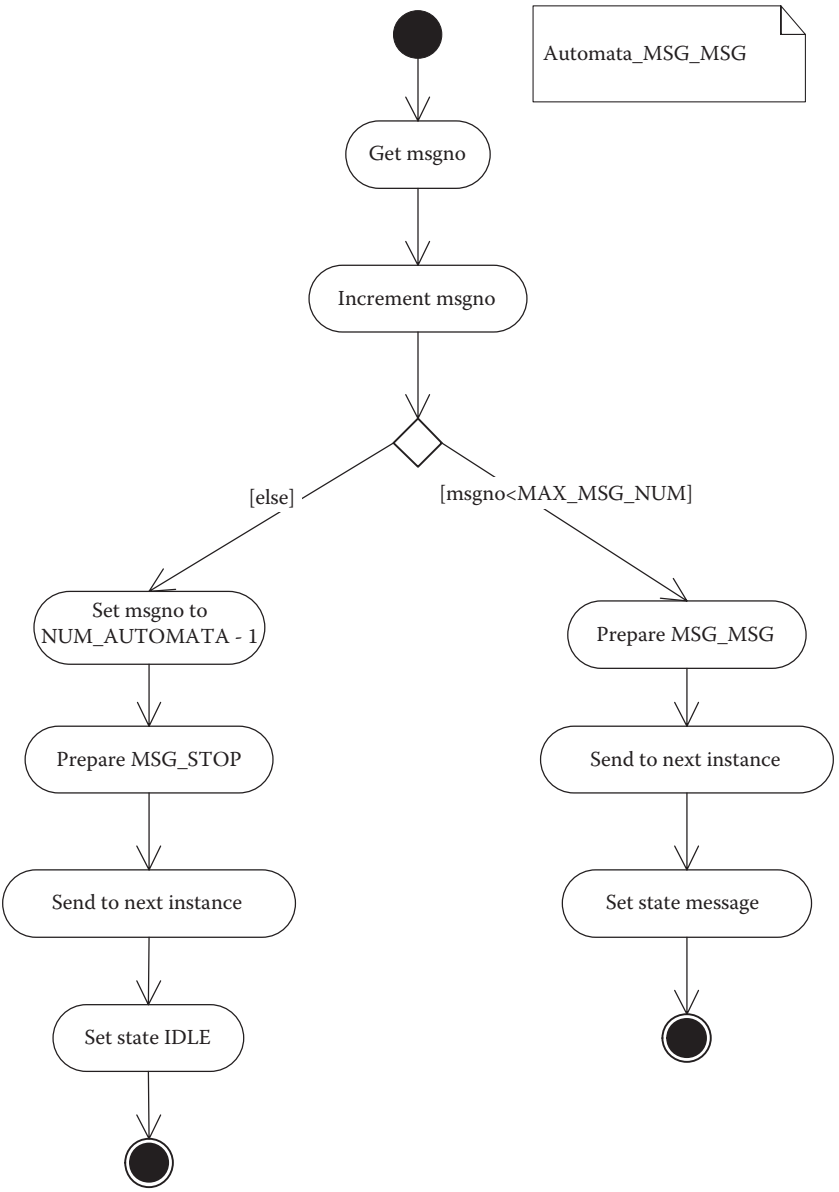


FIGURE 6.11
Activity diagram for the operation *Automata_MSG_MSG*.

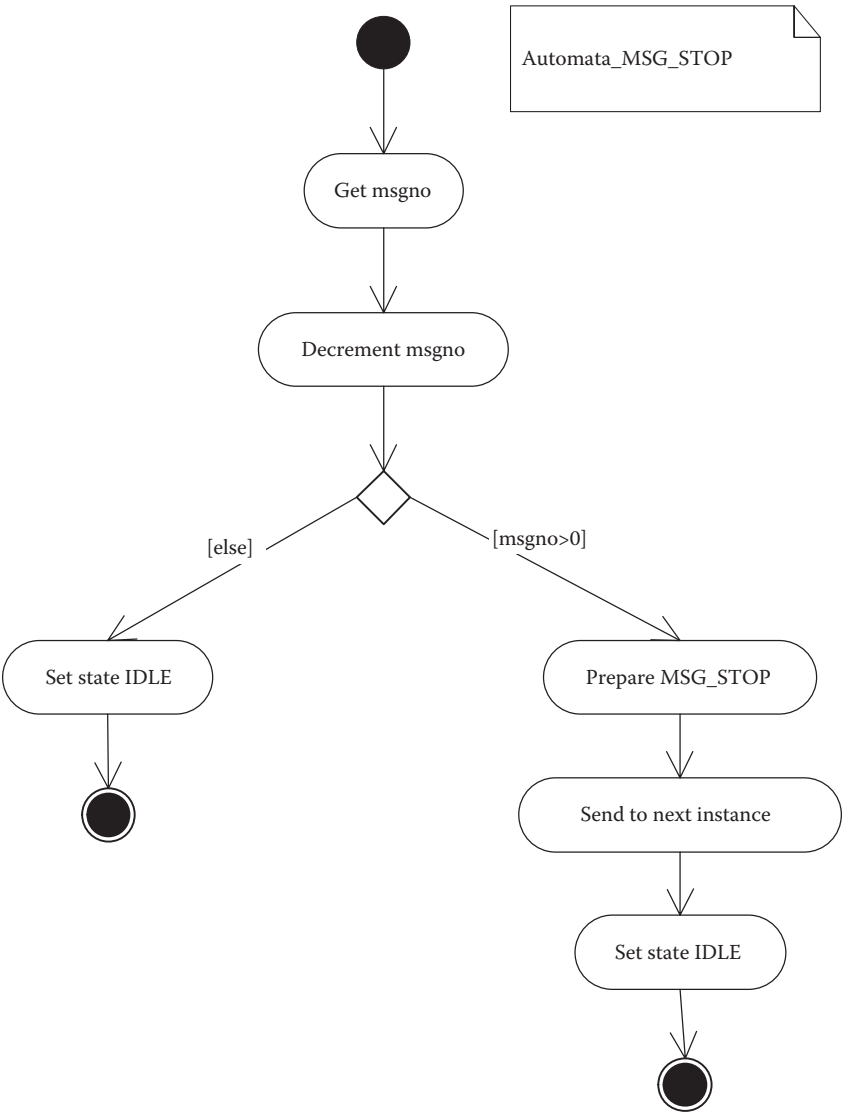


FIGURE 6.12
Activity diagram for the operation *Automata_MSG_STOP*.

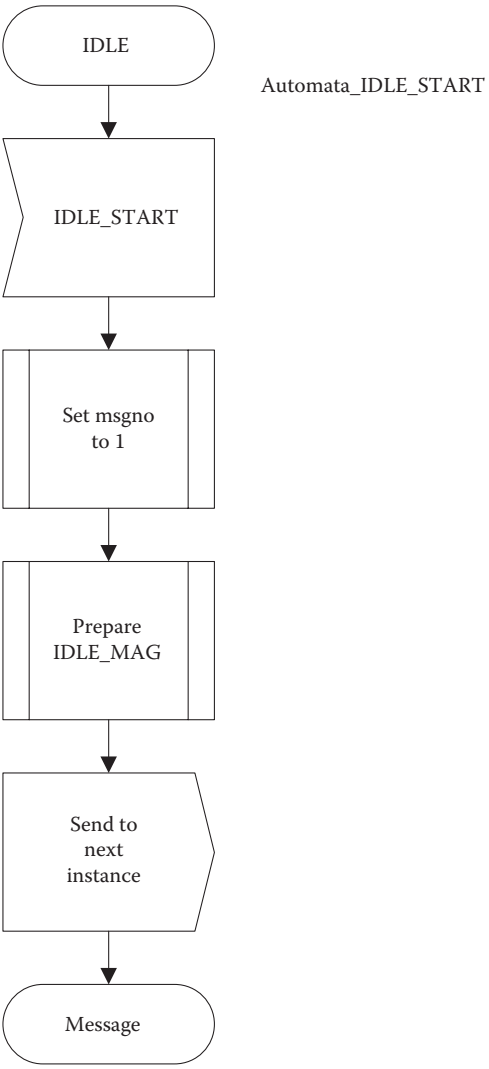


FIGURE 6.13
SDL diagram for the transition *Automata_IDLE_START*.

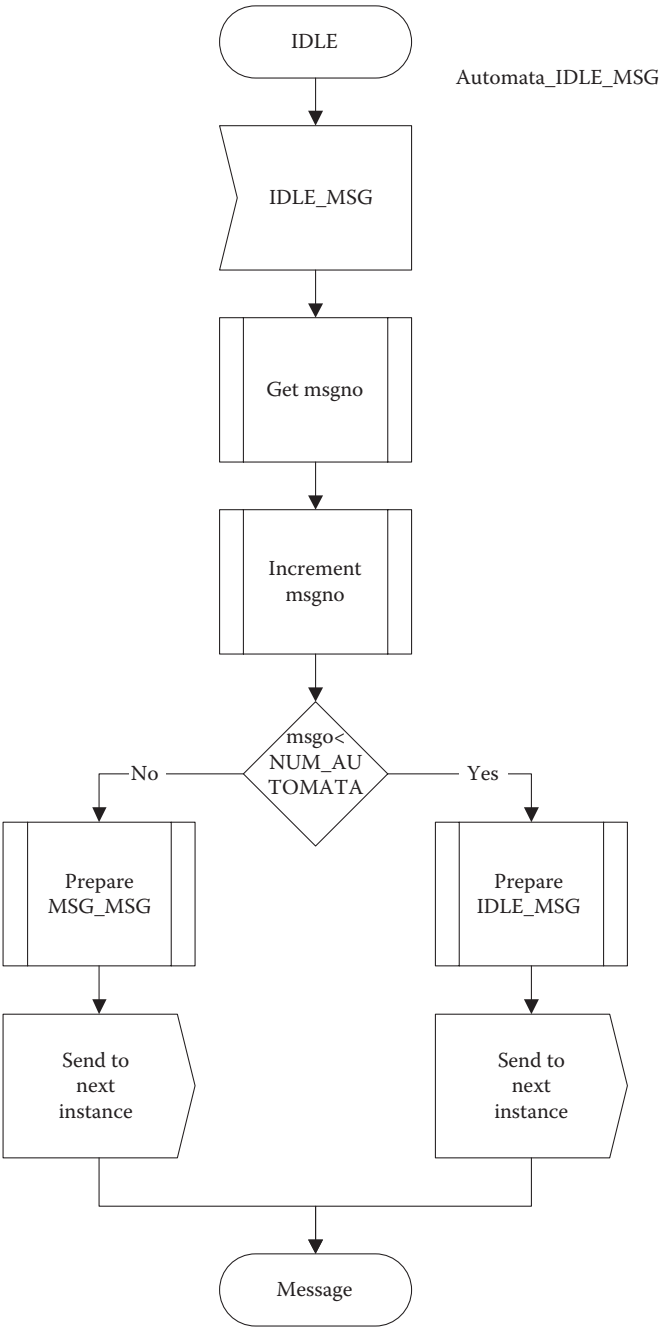


FIGURE 6.14
SDL diagram for the transition *Automata_IDLE_MSG*.

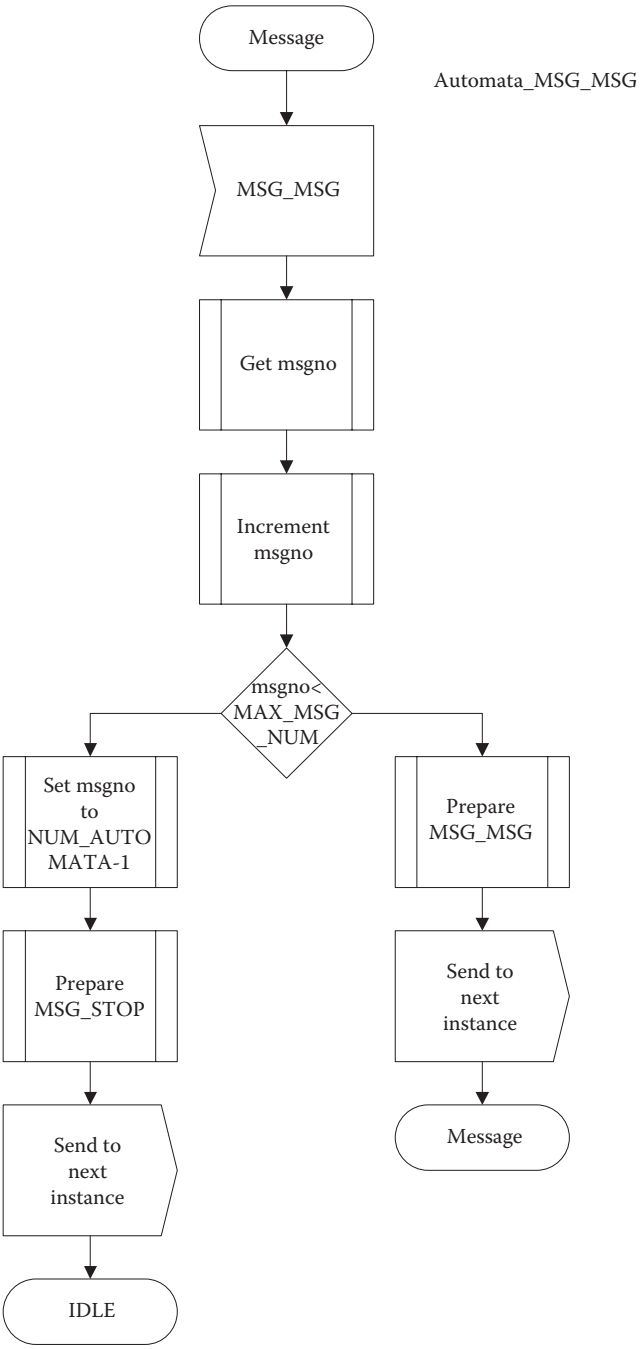


FIGURE 6.15
SDL diagram for the transition *Automata_MSG_MSG*.

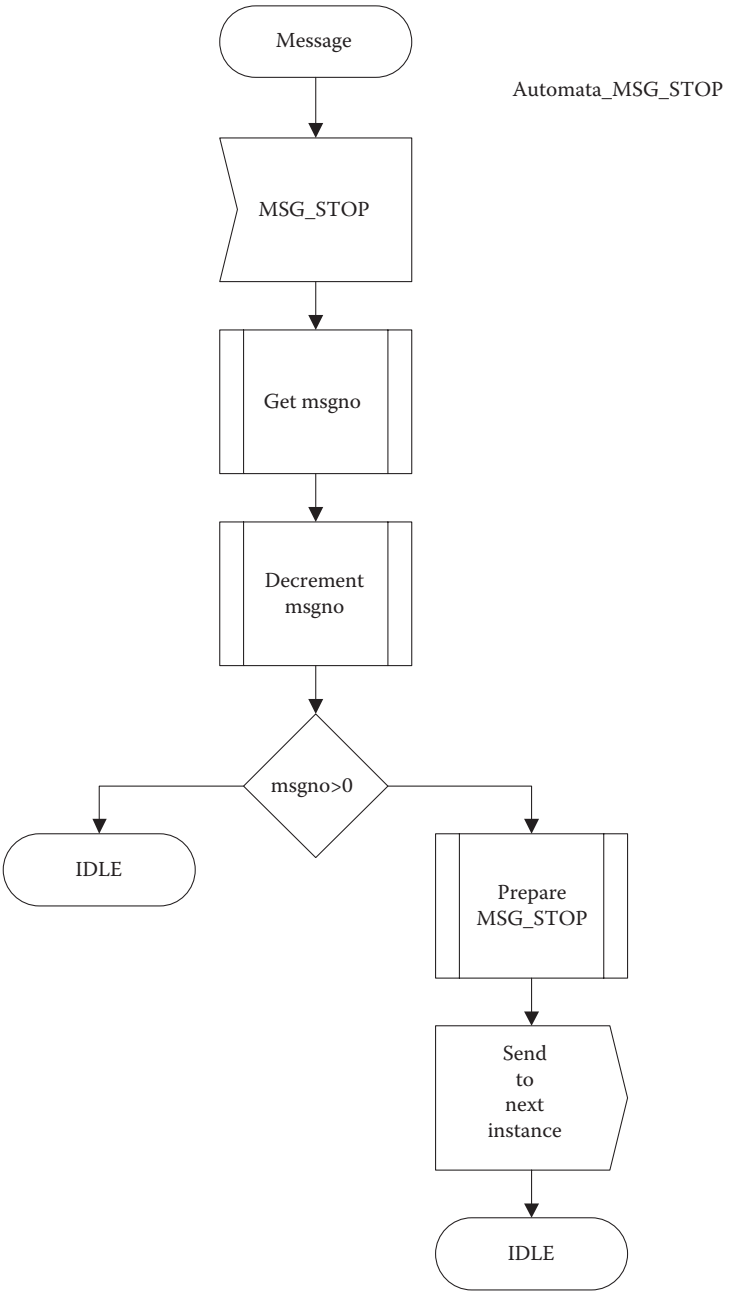


FIGURE 6.16
SDL diagram for the transition *Automata_MSG_STOP*.

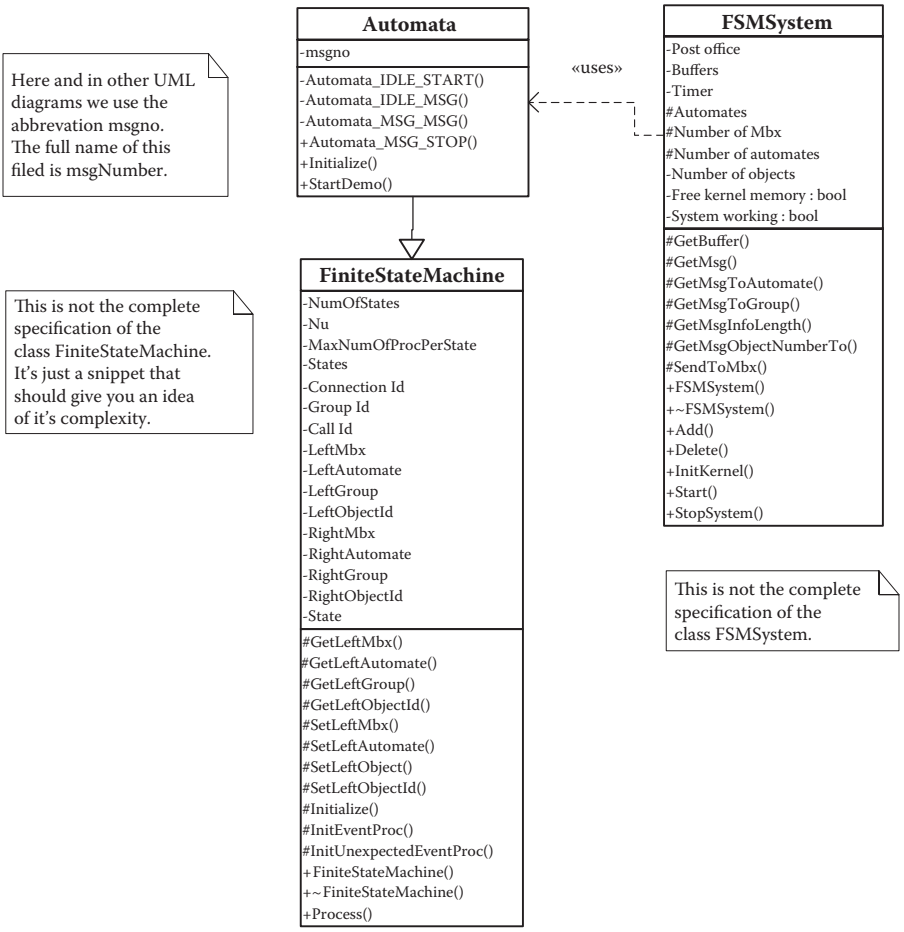


FIGURE 6.17
Class diagram for the example with three automata instances.

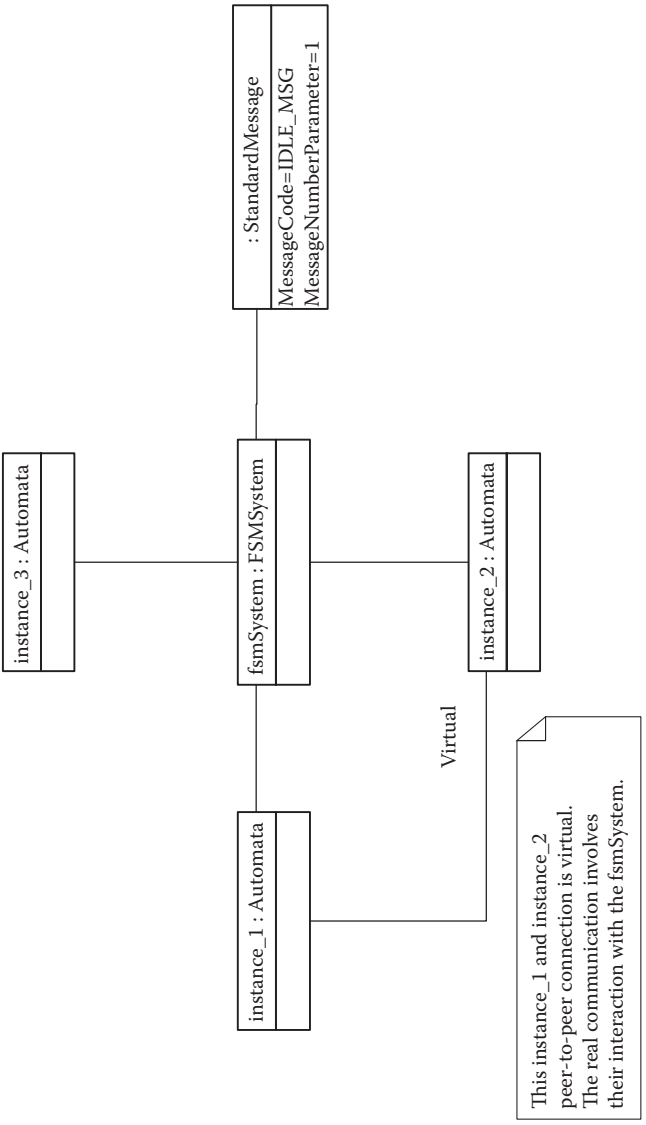


FIGURE 6.18
Object diagram for the example with three automata instances.

The program project in this example comprises the files *Automata.h*, *Automata.cpp*, *Constants.h*, *Main.cpp*, and the FSM Library (see the corresponding component diagram in Figure 6.19). Building this project in Microsoft® Visual Studio 6.0 yields a single executable, which is executed on a single PC machine (see the corresponding deployment diagram in Figure 6.20).

The rest of this section is devoted to the program implementation of the previous models. The content of the corresponding program files is as follows.

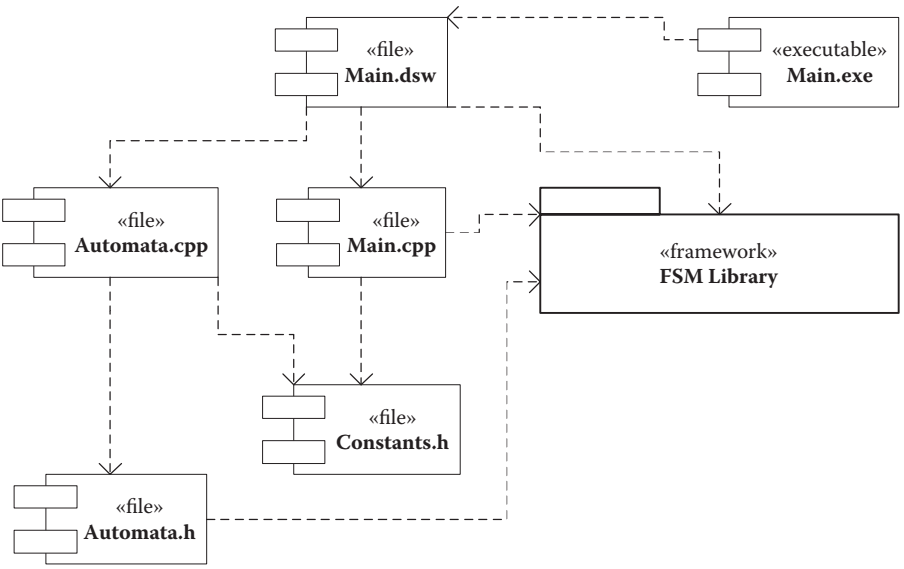


FIGURE 6.19
Component diagram for the example with three automata instances.

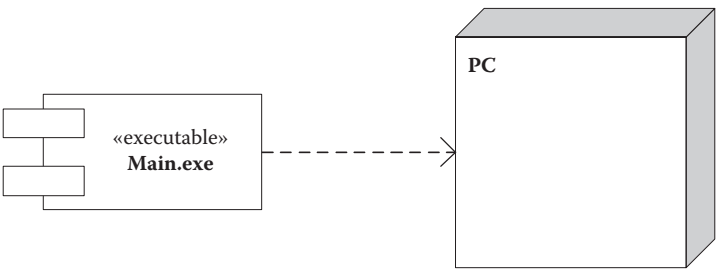


FIGURE 6.20
Deployment diagram for the example with three automata instances.

File *Automata.h*:

```
#ifndef __AUTOMATA__
#define __AUTOMATA__
#include <stdio.h>
#include "stdlib.h"
#include "kernel\fsm.h"
#include "kernel\errorObject.h"
#include "Constants.h"

class Automata: public FiniteStateMachine {
 private:
  StandardMessage StandardMsgCoding;
  MessageInterface *GetMessageInterface(uint32 id);

  void SetDefaultHeader(uint8 infoCoding);
  uint8 GetMbxId();
  uint8 GetAutomata();
  void SetDefaultFSMData();
  void NoFreeInstances();

  uint8 text[20];
  uint32 msgNumber;
  uint32 idToMsg;

  // State transition functions for the state IDLE
  void Automata_IDLE_START();
  void Automata_IDLE_MSG();
  // State transition functions for the state MSG
  void Automata_MSG_MSG();
  void Automata_MSG_STOP();
  // Unexpected event handlers for the states IDLE and  MSG
  void Automata_UNEXPECTED_IDLE();
  void Automata_UNEXPECTED_MSG();

 public:
  Automata();
  ~Automata(){};

 void Initialize();
 void StartDemo();
};
#endif
```

The file *Automata.h* contains a declaration of the class *Automata* derived from the class *FiniteStateMachine*. This declaration has its private and public parts. The private field members are the message interface object *StandardMsgCoding*, the text work area *text*, the message sequence number *msgNumber*, and the identification of the message destination automata *idToMsg*.

The common private function members are the following functions:

- *GetMessageInterface*: returns the message interface object
- *SetDefaultHeader*: sets the message header in accordance with the specified information coding
- *GetMbxId*: returns the identification of the mailbox assigned to this automata type

- *GetAutomata*: returns the identification of this automata type
- *SetDefaultFSMData*: sets the data specific for this automata type (*msgNumber* and *idToMsg*)
- *NoFreeInstances*: handles the situation when no more free instances of this type are found

The application-specific private function members are the following state transition functions:

- *Automata_IDLE_START*: handles the message *IDLE_START* in the state *IDLE*
- *Automata_IDLE_MSG*: handles the message *IDLE_MSG* in the state *IDLE*
- *Automata_MSG_MSG*: handles the message *MSG_MSG* in the state *MESSAGE*
- Automata_*MSG_STOP*: handles the message *MSG_STOP* in the state *MESSAGE*
- *Automata_UNEXPECTED_IDLE*: handles unexpected messages in the state *IDLE*
- Automata_*UNEXPECTED_MSG*: handles unexpected messages in the state *MESSAGE*

The public function members are the class constructor, the class destructor, the initialization function *Initialize*, and the startup function *StartDemo*.

File *Automata.cpp*:

```
#include "kernel/LogFile.h"
#include "Automata.h"

Automata::Automata() : FiniteStateMachine(
 0, // uint16 numOfTimers = DEFAULT_TIMER_NO,
 2, // uint16 numOfState = DEFAULT_STATE_NO,
 3) // uint16 maxNumOfProceduresPerState = DEFAULT_PROCEDURE_NO_PER_STATE
 {
 SetDefaultFSMData();
 }

// This function returns the pointer to the object that governs the
// message information coding (the pointer to the message interface).
// This automata instance works only with the standard messages
// (ID 0x00). If the caller specifies another type of coding,
// this function throws the exception TErrorObject. The message
// interface is defined in Automata.h
MessageInterface *Automata::GetMessageInterface(uint32 id){
 switch(id){
  case 0x00:
   return &StandardMsgCoding;
 }
```

```
 throw TErrorObject(__LINE__,__FILE__,0x01010400);
}

// This function fills in the message header.
void Automata::SetDefaultHeader(uint8 infoCoding){
 SetMsgInfoCoding(infoCoding);
 SetMessageFromData();
}

// This function returns the identification of the mailbox that is
// assigned to this automata type.
uint8 Automata::GetMbxId(){
 return MBX_AUTOMATA_ID;
}

// This function returns the identification of this automata type.
uint8 Automata::GetAutomata(){
 return FSM_TYPE_AUTOMATA;
}

// This function initializes the data specific to individual
// instance of this automata type.
void Automata::SetDefaultFSMData(){
 msgNumber = 0;
 idToMsg  = INVALID_32;
}
// This function is called if there are no free instances of this
// automata type. If the programmer wants to use this option, they must
// add the first automata instance of this type to the parameter
// useFreeList of the function Add set to true. In this example, it
// is empty. In real applications, the programmer should provide
// some recovery mechanism, such as overload protection or restart.
void Automata::NoFreeInstances(){
}

// This function initializes the state transition functions and the
// timers that are used by this automata type. This function is
// called implicitly by the function Add, which is responsible for
// adding individual automata instances to the FSM system.
// Each state transition function is separately declared and defined.
void Automata::Initialize(){
 // Here the programmer does the following initializations:
 // InitEventProc(uint8 state, uint16 event, PROC_FUN_PTR fun);
 // InitUnexpectedEventProc(uint8 state, PROC_FUN_PTR fun);
 // InitTimerBlock(uint16 timerId, uint32 timerCount, uint16 signalId);
 InitEventProc(IDLE,IDLE_START,(PROC_FUN_PTR)
  &Automata::Automata_IDLE_START);
 InitEventProc(IDLE,IDLE_MSG,(PROC_FUN_PTR)
  &Automata::Automata_IDLE_MSG);

 InitEventProc(MESSAGE,MSG_MSG,(PROC_FUN_PTR)
  &Automata::Automata_MSG_MSG);
 InitEventProc(MESSAGE,MSG_STOP,(PROC_FUN_PTR)
  &Automata::Automata_MSG_STOP);

 InitUnexpectedEventProc(IDLE,(PROC_FUN_PTR)
  &Automata::Automata_UNEXPECTED_IDLE);
 InitUnexpectedEventProc(MESSAGE,(PROC_FUN_PTR)
  &Automata::Automata_UNEXPECTED_MSG);
}

// State transition functions for the state IDLE.
void Automata::Automata_IDLE_START(){
 msgNumber = 1;
```

```
 idToMsg  = GetObjectId()+1;

 // Round Robin message transfer among automata instances 0-2
 if(idToMsg == 3)
  idToMsg = 0;

 // The automata instance prepares and sends the message,
 // and changes its state to MESSAGE.
 PrepareNewMessage(0x00,IDLE_MSG);

 char text[] = "THIS IS THE FIRST MESSAGE";
 AddParam(PARAM_TEXT,strlen(text),(unsigned char *)text);
 AddParamDWord(COUNT,msgNumber);

 SetMsgToAutomata(FSM_TYPE_AUTOMATA);
 SetMsgToGroup(INVALID_08);
 SetMsgObjectNumberTo(idToMsg);
 SendMessage(MBX_AUTOMATA_ID);
 SetState(MESSAGE);
}

void Automata::Automata_IDLE_MSG(){
 idToMsg = GetObjectId()+1;

 // Round Robin message transfer among automata instances 0-2
 if((idToMsg == 3)
  idToMsg = 0;
 // Get parameters from the message
 unsigned char *tmp;
 tmp = GetParam(PARAM_TEXT);
 assert(tmp);
 memcpy(text,tmp+2,*(tmp+1));
 memset(text+(*(tmp+1)),0x00,1); // make the string
 GetParamDWord(COUNT,msgNumber);

 // Round Robin - this instance receives the message from the previous one
 uint32 idFromMsg = GetObjectId()-1;
 if(idFromMsg == -1)
  idFromMsg = 2;

 printf("Text received: %s\n from automata:%u \n",text,idFromMsg);

 // If the message sequence number is less than NUM_AUTOMATA,
 // send IDLE_MSG. If not, send MSG_MSG.
 msgNumber++;
 if(msgNumber < NUM_AUTOMATA){
  // Prepare and send the message.
  // Change automata state to MESSAGE.
  PrepareNewMessage(0x00,IDLE_MSG);

  char text[] = "THIS IS THE SECOND MESSAGE";
  AddParam(PARAM_TEXT,strlen(text),(unsigned char *)text);
  AddParamDWord(COUNT,msgNumber);

  SetMsgToAutomata(FSM_TYPE_AUTOMATA);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(idToMsg);
  SendMessage(MBX_AUTOMATA_ID);
 }
 else {
  // Prepare and send the message.
  // Change automata state to MESSAGE.
  PrepareNewMessage(0x00,MSG_MSG);
  AddParamDWord(COUNT,msgNumber);
```

```
  SetMsgToAutomata(FSM_TYPE_AUTOMATA);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(idToMsg);
  SendMessage(MBX_AUTOMATA_ID);
 }
 SetState(MESSAGE);
}

void Automata::Automata_MSG_MSG(){
 GetParamDWord(COUNT,msgNumber);
 msgNumber++;
 if(msgNumber < MAX_MSG_NUM){
  // Forward the message to the next automata   instance.
  PrepareNewMessage(0x00,MSG_MSG);
  AddParamDWord(COUNT,msgNumber);
  SetMsgToAutomata(FSM_TYPE_AUTOMATA);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(idToMsg);
  SendMessage(MBX_AUTOMAT_ID);
 }
 else {
  printf("Stop automata:%with message:%u\n",GetObjectId(),msgNumber);

  // Prepare and send the message.
  // Change automata state to IDLE.
  PrepareNewMessage(0x00,MSG_STOP);
  AddParamDWord(COUNT,NUM_AUTOMATA-1);
  SetMsgToAutomata(FSM_TYPE_AUTOMATA);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(idToMsg);
  SendMessage(MBX_AUTOMATA_ID);
  SetState(IDLE);
 }
}

void Automata::Automata_MSG_STOP(){
 printf("Stop automata instance: %u\n",GetObjectId());

 GetParamDWord(COUNT,msgNumber);
 msgNumber——;
 if(msgNumber > 0){
  // Prepare and send the message.
  // Change automata state to IDLE.
  PrepareNewMessage(0x00,MSG_STOP);
  AddParamDWord(COUNT,msgNumber);
  SetMsgToAutomata(FSM_TYPE_AUTOMATA);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(idToMsg);
  SendMessage(MBX_AUTOMATA_ID);
 }
 SetState(IDLE);
}

void Automata::Automata_UNEXPECTED_IDLE(){
 printf("Unexpected message in the state IDLE \n");
}

void Automata::Automata_UNEXPECTED_MSG(){
 printf("Unexpected message in the state MESSAGE \n");
}

void Automata::StartDemo(){
 uint8 *msg = GetBuffer(MSG_HEADER_LENGTH);
```

```
 SetMsgFromAutomata(FSM_TYPE_AUTOMATA,msg);
 SetMsgFromGroup(INVALID_08,msg);
 SetMsgObjectNumberFrom(0,msg);

 SetMsgToAutomata(FSM_TYPE_AUTOMATA,msg);
 SetMsgToGroup(INVALID_08,msg);
 SetMsgObjectNumberTo(0,msg);

 SetMsgInfoCoding(0,msg); // 0 = StandardMessage
 SetMsgCode(IDLE_START,msg);
 SetMsgInfoLength(0,msg);
 SendMessage(MBX_AUTOMATA_ID,msg);
}
```

The file *Automata.cpp* contains the definition of the class *Automata*. This definition starts with the class constructor that first calls the base class constructor specifying no timers, two states, and the maximum of three state transitions per state for this automata type. After that, the constructor calls the function *SetDefaultFSMData*, which sets the data specific for this automata type.

The function *GetMessageInterface* returns the pointer to the message interface object for the given type of information coding. This class operates with only standard messages (the corresponding ID is 0x00). If the caller of this function specifies the identification of the standard message as its parameter, the function returns the pointer to the object *StandardMsgCoding*. If the caller specifies some other message type, this function throws the exception *TErrorObject*.

The function *SetDefaultHeader* sets the message information coding by calling the function *SetMsgInfoCoding* and the automata specific data by calling the function *SetMessageFromData*. The function *GetMbxId* returns the value *MBX_AUTOMATA_ID* as the identification of the mailbox assigned to this automata type. The function *GetAutomata* returns the value *FSM_TYPE_AUTOMATA* as the identification of this automata type. The function *SetDefaultFSMData* sets the field *msgNumber* to the value 0 and the field *idToMsg* to the value *INVALID_32*. The function *NoFreeInstances* is empty in this simple example. In real-world projects, it would be used to trigger some higher-level protection or recovery mechanism.

The function *Initialize* defines the event handlers by calling the function *InitEventProc* and the unexpected event handlers by calling the function *InitUnexpectedEventProc*. More precisely, this function defines the event handlers for the messages *IDLE_START* and *IDLE_MSG* in the state *IDLE*, and for the messages *MSG_MSG* and *MSG_STOP* in the state *MESSAGE*. It also defines the handlers for unexpected messages in both states.

The function *Automata_IDLE_START* handles the message *IDLE_START* in the state *IDLE*. First, it sets the message sequence number *msgNumber* to the value 1. It then determines the identification of the destination automata instance by incrementing its own identification by modulo 3. (This means that the destination of the messages created and sent by *instance_0* is *instance_1*, the destination for *instance_1* is *instance_2*, and the destination for *instance_2* is *instance_0*.) Next, this function prepares and sends the message, "THIS IS THE FIRST MESSAGE". At the end, it performs the state transition from

IDLE to *MESSAGE* by calling the function *SetState* and specifying the value *MESSAGE* as its parameter.

The function *Automata_IDLE_MSG* handles the message *IDLE_MSG* in the state *IDLE*. First, it determines the identifications of the source and destination automata instances for the received message and prints them to the monitor. It then increments the message sequence numbers and checks if they are less than the number of communicating automata instances *NUM_AUTOMATA* (value 3). If yes, the function prepares and sends the message *IDLE_MSG* with the text, "THIS IS THE SECOND MESSAGE". If not, the function prepares and sends the message *MSG_MSG* without any text. In both cases, it sets the current state of the automata instance to the value *MESSAGE*.

The function *Automata_MSG_MSG* handles the message *MSG_MSG* in the state *MESSAGE*. First, it gets the message sequence number from the received message and increments that number. It then checks if the new value of the message sequence number has reached the given limit. If not, this function prepares and sends the message *MSG_MSG* to the next automata instance in the chain. If it has, this function prepares and sends the message *MSG_STOP* to the next automata instance in the chain, and sets the current state of this automata instance to *IDLE*.

The function *Automata_MSG_STOP* handles the message *MSG_STOP* in the state *MESSAGE*. First, it decrements the message sequence number and checks its new value. If the value is positive, the function prepares and sends the message *MSG_STOP* to the next automata instance in the chain, and sets the current state of this automata instance to *IDLE*.

The unexpected event handlers in this example just print the warning messages. In real applications, these functions would trigger some higher-level recovery mechanisms. The function *StartDemo* creates the first message in the system. It fills in its header as if the automata instance with the identification 0 had sent that message to itself, and sends the message to the mailbox assigned to this automata type.

File *Constants.h*:

```
// FSM
#define FSM_TYPE_AUTOMATA 0

// MBX
#define MBX_AUTOMATA_ID 0

#define MAX_MSG_NUM 10
#define NUM_AUTOMATA 3
#define COUNT 1
#define PARAM_TEXT 2

enum AutomataStates{
 IDLE = 0,
 MESSAGE,
};
```

```
enum Messages{
 IDLE_START = 0,
 IDLE_MSG,
 MSG_MSG,
 MSG_STOP
};
```

The file *Constants.h* first defines general symbolic constants. The identification of the automata type *FSM_TYPE_AUTOMATA* is assigned the value 0, the identification of the mailbox related to the automata type *MBX_AUTOMATA_ID* is assigned the value 0, the maximal message sequence number *MAX_MSG_NUM* is assigned the value 10, the number of automata instances of this type *NUM_AUTOMATA* is assigned the value 3, the identification of the message parameter that contains the messages sequence number *COUNT* is assigned the value 1, and the identification of the message parameter that contains the text *PARAM_TEXT* is assigned the value 2.

Next, the identifications of the individual states of this automata type are enumerated. The identification of the state *IDLE* is assigned the value 0 and the identification of the state *MESSAGES* is assigned the value 1. Finally, the identifications of various message types (message codes) are enumerated. The message types are named as *IDLE_START*, *IDLE_MSG*, *MSG_MSG*, and *MSG_STOP*. These symbols are assigned the values 0, 1, 2, and 3, respectively.

File *Main.cpp*:

```
#include "conio.h"
#include "Kernel/fsmsystem.h"
#include "Kernel/LogFile.h"
#include "Automata.h"

// Assume the following.
// The FSM system hosts a single automata type.
// The FSM system uses a single mailbox for the message exchange.
// Create the FSM system.
FSMSystem fsmSystem(1,1);

// Create three instances of the class Automata.
Automata instance_1, instance_2, instance_3;

// FSM system thread
DWORD WINAPI ThreadFunction(void* dummy){
 uint32 buffersCount[3] = {5,3,2};
 uint32 buffersLength[3] = {128,256,512};
 uint8 buffClassNo = 3;
 // Initialize the FSM system.
 printf("Initialize the FSM system... \n");
 fsmSystem.Add(&instance_1,FSM_TYPE_AUTOMATA,3,false);
 fsmSystem.Add(&instance_2,FSM_TYPE_AUTOMATA);
 fsmSystem.Add(&instance_3,FSM_TYPE_AUTOMATA);

 fsmSystem.InitKernel(buffClassNo,buffersCount,buffersLength,1);
```

```
 LogFile lf("log.log", "log.ini");
 LogAutomataNew::SetLogInterface(&lf);

 // Start the FSM system.
 printf("Start the FSM system... \n");
 try {
   fsmSystem.Start();
 }
 catch(...) {
   OutputDebugString("Exception — stop the FSM    system...\n");
   return 0;
 }
 OutputDebugString("The end of the operation.\n");
 return 0;
}

void main(int argc,char* argv[]){
 DWORD threadID;
 bool end = false;
 char ret;

 // Start the FSM system thread.
 HANDLE hTemp =  CreateThread(NULL,0,ThreadFunction,NULL,0,&threadID);
 Sleep(100);

 // Program works until the character 'Q' or 'q' is         pressed.
 while(!end) {
  if(_kbhit()) {
   ret = _getch();
   switch(ret) {
    case 'Q':
    case 'q':
     fsmSystem.StopSystem();
     end = true;
     Sleep(100);
     break;
    case 'S':
    case 's':
     instance_1.StartDemo();
     break;
    default:
     break;
   }
  }
 }
 CloseHandle(hTemp);
 printf("The end. \n");
}
```

The file *Main.cpp* starts with the instantiation of the class *FSMSystem* by calling its constructor. The parameters used in this call specify that an instance of the *FSMSystem*, named *fsmSystem*, will include a single automata type, and this automata type will use a single mailbox. Next, three instances of the class *Automata* are made, namely, *instance_1*, *instance_2*, and *instance_3*. Additionally, this file contains the definitions of the FSM system thread function *ThreadFunction* and the function *main*.

The function *ThreadFunction* first prepares the data needed to define three buffer types. The sizes and quantities of these buffers are five at 128 bytes, three at 256 bytes, and two at 512 bytes. Next, three automata

instances are added to *fsmSystem*. Note that the fourth parameter of the first call to the function *Add* is set to the value *false*, which means that these three instances are to be used as three distinctive instances, rather than as a pool of instances of the same type. After that, this function initializes the kernel by calling the function *InitKernel*, defines and sets the logging interface by calling the function *SetLogInterface*, and starts the *fsmSystem* by calling its function *Start*.

The function *main* starts the FSM system thread (which executes the function *ThreadFunction*) and suspends itself for 100 ms. After that, it just waits for the character 'Q' or 'q' to be pressed and to subsequently terminate the program.

6.10 A Simple Example with Network-Aware Automata Instances

This section shows how the programmer can construct FSM systems with TCP support that is able to communicate over the TCP/IP network, and how they can add individual, network-aware automata instances to it. Normally, the programmer creates the FSM system with TCP support by instantiating the class *FSMSystemWithTCP*. Alternately, network-aware automata types are normally derived from the base class *NetFSM*. Of course, network-aware automata instances of a given type are then created simply by instantiating that automata type.

This example is very similar to the previous one. Actually, it has been created from it with a few rather simple modifications. Only one instance of the given automata type is added to the FSM system (now with TCP/IP support). This automata instance has a trivial task of exchanging the given number of messages with its peers in the remote FSM system. The main difference is that the whole program is instantiated twice. These program instances run as two separate processes that communicate over the TCP/IP protocol stack (see the corresponding collaboration diagram in Figure 6.21).

At the beginning, as in the previous example, the main thread calls the function *StartDemo* of *instance_1*, which, in turn, sends itself asynchronously the message *IDLE_START*. Upon reception of this message, *instance_1* sends the message *IDLE_MSG* to its peer *instance_1* that resides at the remote FSM system. These two automata instances, local and remote, then exchange nine *MSG_MSG* messages (the last *MSG_MSG* message is not shown in the figure). At the end of the communication, the local instance sends the message *MSG_STOP* to the remote instance (not shown in the figure). The corresponding sequence diagram is shown in Figure 6.22. This diagram shows all the messages.

FIGURE 6.21
Collaboration diagram for the example with network-aware automata.

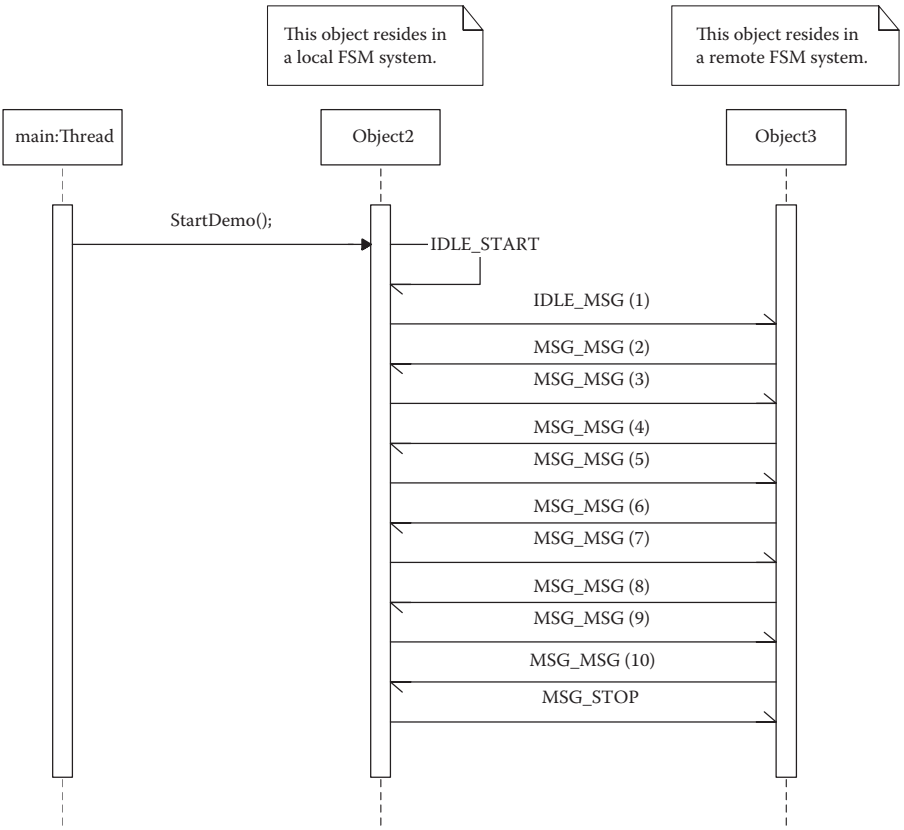


FIGURE 6.22
Sequence diagram for the example with network-aware automata.

The statechart diagram that describes the behavior of an individual automata instance is again organized into two hierarchical levels. The top level is exactly the same as the one shown in Figure 6.4. The composite states *Automata_IDLE_START*, *Automata_IDLE_MSG*, *Automata_MSG_MSG*, and *Automata_MSG_STOP* are a little simpler in this example and are shown in Figures 6.23 through 6.26, respectively.

The program code given in this example assumes that both processes run on the same machine whose IP address is 192.168.0.57. To get this code running on another machine, the reader should change this parameter accordingly. If the reader wants to experiment on two different machines, they must set this parameter to the IP addresses of those machines (see the corresponding deployment diagram shown in Figure 6.27).

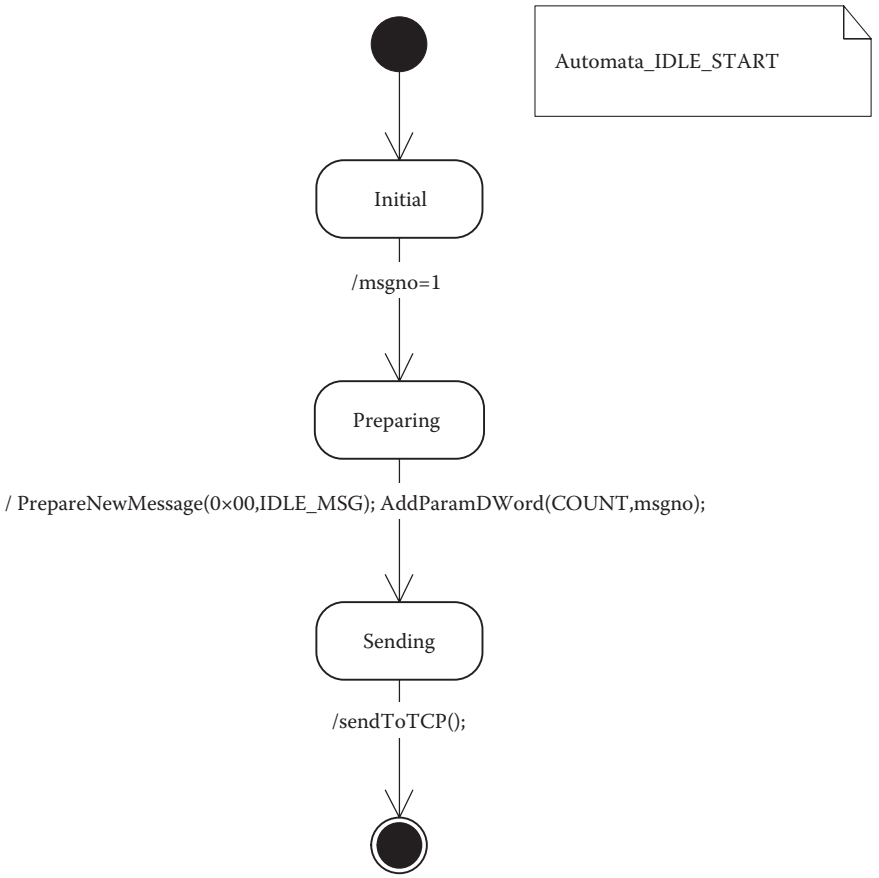


FIGURE 6.23
Composite state *Automata_IDLE_START*.

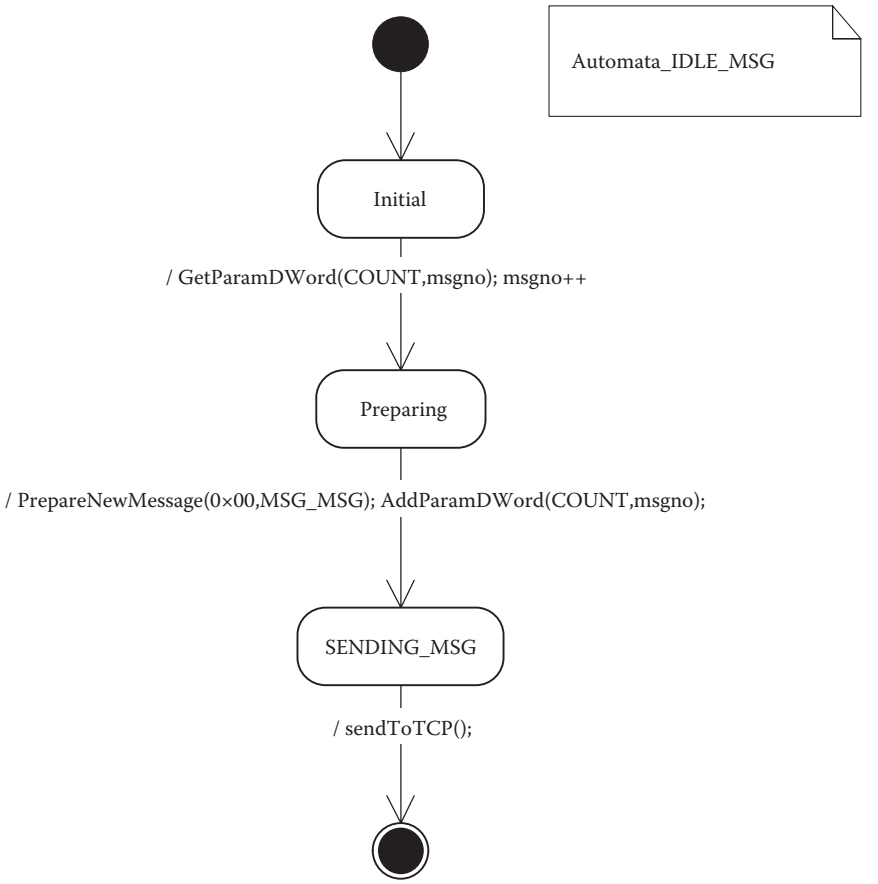


FIGURE 6.24
Composite state *Automata_IDLE_MSG*.

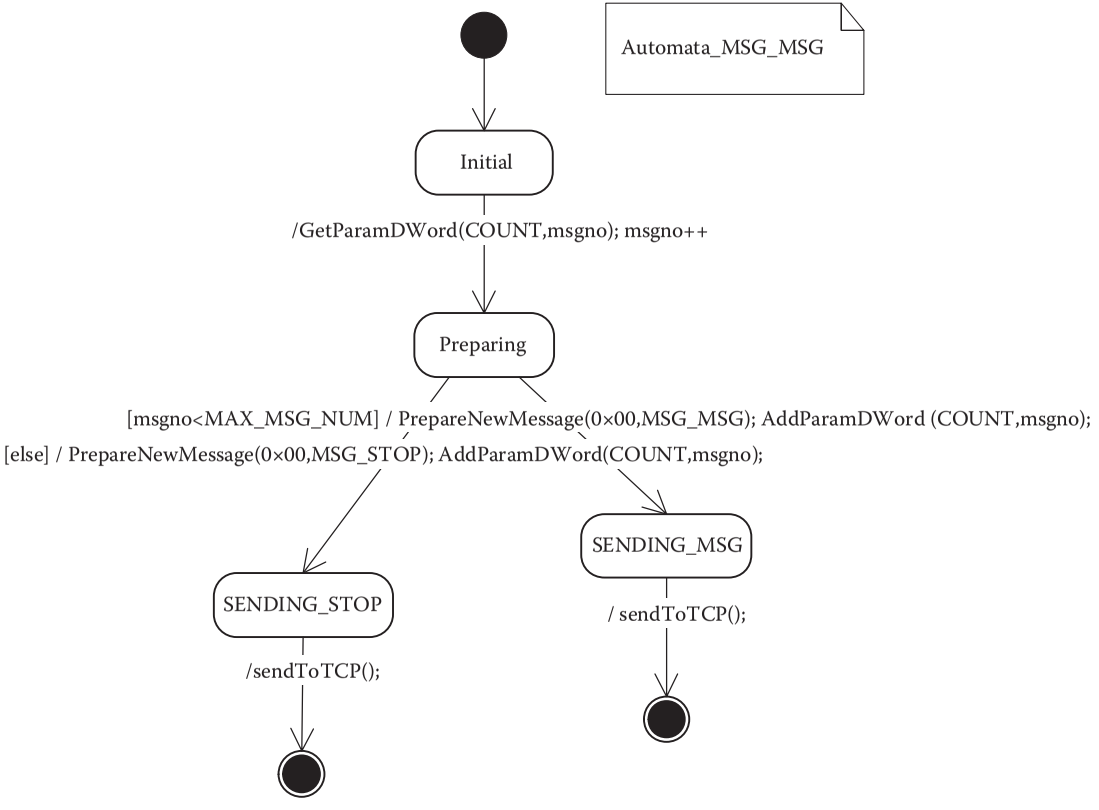


FIGURE 6.25
Composite state *Automata_MSG_MSG*.

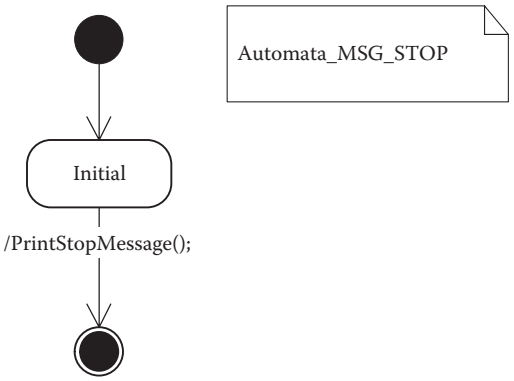


FIGURE 6.26
Composite state *Automata_MSG_STOP.*

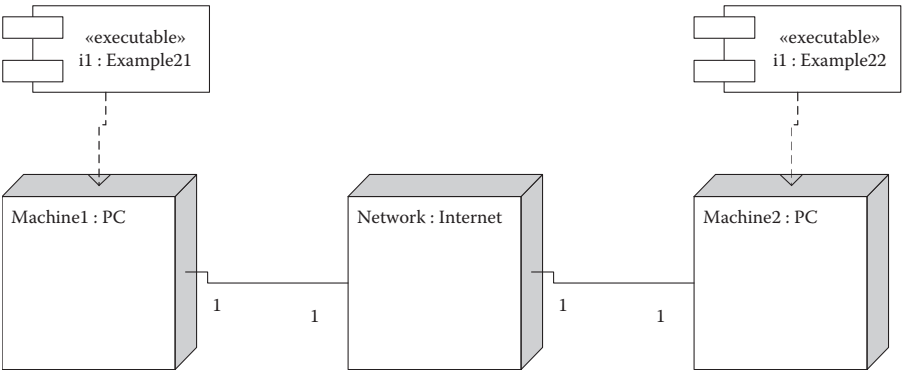


FIGURE 6.27
Deployment diagram for the example with network-aware automata.

Before proceeding further, studying the previous example first is strongly recommended. The content of the program files are as follows:

File *NetAutomata.h*:

```
#ifndef __NET_AUTOMATA__
#define __NET_AUTOMATA__
#include <stdio.h>
#include "stdlib.h"
#include "kernel\NetFSM.h"
#include "kernel\errorObject.h"
#include "Constants.h"

class NetAutomata: public NetFSM {
 private:
  // NetFSM
  uint16 convertNetToFSMMessage();
```

```
  void convertFSMToNetMessage();
  uint8 getProtocolInfoCoding();
  // FSM
  StandardMessage StandardMsgCoding;
  MessageInterface *GetMessageInterface(uint32 id);
  void SetDefaultHeader(uint8 infoCoding);
  uint8 GetMbxId();
  uint8 GetAutomata();
  void SetDefaultFSMData();
  void NoFreeInstances();

  uint8 text[20];
  uint32 msgNumber;
  uint32 idToMsg;

  // State transition functions for the state IDLE
  void NetAutomata_IDLE_START();
  void NetAutomata_IDLE_MSG();
  // State MSG
  void NetAutomata_MSG_MSG();
  void NetAutomata_MSG_STOP();
  // Unexpected messages in states IDLE and MSG
  void NetAutomata_UNEXPECTED_IDLE();
  void NetAutomata_UNEXPECTED_MSG();

 public:
  NetAutomata();
  ~NetAutomata(){};
  void Initialize();
  void StartDemo();
   };
#endif
```

The file *NetAutomata.h* contains the declaration of the class *NetAutomata* derived from the class *NetFSM*. This declaration has its private and public parts. The private field members are the message interface object *StandardMsgCoding*, the text work area *text*, the message sequence number *msgNumber*, and the identification of the automata instance *idToMsg*, which is the message destination.

The private function members specific to the class *NetFSM* are the following functions:

- *convertNetToFSMMessage*: converts the external message format into the internal message format appropriate for communication within the FSM system
- *convertFSMToNetMessage*: converts the internal message format into the external message format appropriate for the transmission over the TCP/IP network
- *getProtocolInfoCoding*: returns the identification of the type of the external message coding

The private function members specific to the class *FinteStateMachine* are the following functions:

- *GetMessageInterface*: returns the message interface object

- *SetDefaultHeader*: sets the message header according to the specified information coding
- *GetMbxId*: returns the identification of the mailbox that is assigned to this automata type
- *GetAutomata*: returns the identification of this automata type
- *SetDefaultFSMData*: sets the data specific for this automata type (*msgNumber* and *idToMsg*)
- *NoFreeInstances*: handles the situation when no more free instances of this type are found

The application-specific private function members are the following state transition functions:

- *Automata_IDLE_START*: handles the message *IDLE_START* in the state *IDLE*
- *Automata_IDLE_MSG*: handles the message *IDLE_MSG* in the state *IDLE*
- *Automata_MSG_MSG*: handles the message *MSG_MSG* in the state *MESSAGE*
- *Automata_MSG_STOP*: handles the message *MSG_STOP* in the state *MESSAGE*
- *Automata_UNEXPECTED_IDLE*: handles unexpected messages in the state *IDLE*
- *Automata_UNEXPECTED_MSG*: handles unexpected messages in the state *MESSAGE*

The public function members are the class constructor, the class destructor, the initialization function *Initialize*, and the startup function *StartDemo*.

File *NetAutomata.cpp*:

```
#include "kernel/LogFile.h"
#include "NetAutomata.h"

NetAutomata::NetAutomata() : NetFSM(
 0, // uint16 numOfTimers = DEFAULT_TIMER_NO,
 2, // uint16 numOfState = DEFAULT_STATE_NO,
 3) // uint16 maxNumOfProceduresPerState = DEFAULT_PROCEDURE_NO_PER_STATE
{
 SetDefaultFSMData();
}

// This function returns the pointer to the object that governs the
// message information coding (the pointer to the message interface).
// This automata instance works only with the standard messages
// (ID 0x00). If the caller specifies another type of coding,
// this function throws the exception TErrorObject.
// The message interface is defined in NetAutomata.h
MessageInterface *NetAutomata::GetMessageInterface(uint32 id){
```

```
 switch(id) {
  case 0x00:
   return &StandardMsgCoding;
 }
 throw TErrorObject(__LINE__,__FILE__,0x01010400);
}
// This function fills in the message header.
void NetAutomata::SetDefaultHeader(uint8 infoCoding){
 SetMsgInfoCoding(infoCoding);
 SetMessageFromData();
}

// This function returns the identification of the mailbox that is
// assigned to this automata type.
uint8 NetAutomata::GetMbxId(){
 return MBX_AUTOMATA_ID;
}

// This function returns the identification of this automata type.
uint8 NetAutomata::GetAutomata(){
 return FSM_TYPE_AUTOMATA;
}

// This function initializes the data specific for individual
// instance of this automata type.
void NetAutomata::SetDefaultFSMData(){
 msgNumber = 0;
 idToMsg = INVALID_32;
}

// This function is called if there are no free instances of this
// automata type. If the programmer wants to use this option they must
// add the first automata instance of this type with the parameter
// useFreeList of the function Add set to true. In this example it is
// empty. In real applications the programmer should provide some
// recovery mechanism, such as overload protection or restart.
void NetAutomata::NoFreeInstances(){}

// This function initializes the state transition functions and the
// timers that are used by this automata type. This function is called
// implicitly by the function Add responsible for adding individual
// automata instances to the FSM system.
// Each state transition function is separately declared and defined.
void NetAutomata::Initialize(){
 // Here the programmer does the following  initializations:
 // InitEventProc(uint8 state, uint16 event,  PROC_FUN_PTR fun);
 // InitUnexpectedEventProc(uint8 state, PROC_FUN_PTR  fun);
 // InitTimerBlock(uint16 timerId, uint32 timerCount,  uint16 signalId);

 InitEventProc(IDLE,IDLE_START,(PROC_FUN_PTR)
  &NetAutomata::NetAutomata_IDLE_START);
 InitEventProc(IDLE,IDLE_MSG,(PROC_FUN_PTR)
  &NetAutomata::NetAutomata_IDLE_MSG);

 InitEventProc(MESSAGE,MSG_MSG,(PROC_FUN_PTR)
  &NetAutomata::NetAutomata_MSG_MSG);
 InitEventProc(MESSAGE,MSG_STOP,(PROC_FUN_PTR)
  &NetAutomata::NetAutomata_MSG_STOP);

 InitUnexpectedEventProc(IDLE,(PROC_FUN_PTR)
  &NetAutomata::NetAutomata_UNEXPECTED_IDLE);
 InitUnexpectedEventProc(MESSAGE,(PROC_FUN_PTR)
  &NetAutomata::NetAutomata_UNEXPECTED_MSG);
}
```

```
// State transition functions for the state IDLE.
void NetAutomata::NetAutomata_IDLE_START(){
 msgNumber = 1;
 idToMsg = 0;

 // The automata instance prepares and sends the  message,
 // and changes its state to MESSAGE.
 PrepareNewMessage(0x00,IDLE_MSG);

 char text[] = "THIS IS THE FIRST MESSAGE";
 AddParam(PARAM_TEXT,strlen(text),(unsigned char  *)text);
 AddParamDWord(COUNT,msgNumber);

 SetMsgToAutomata(FSM_TYPE_AUTOMATA);
 SetMsgToGroup(INVALID_08);
 SetMsgObjectNumberTo(idToMsg);
 sendToTCP();
 SetState(MESSAGE);
}

void NetAutomata::NetAutomata_IDLE_MSG(){
 idToMsg = 0;

 // Get parameters from the message
 unsigned char *tmp;
 tmp = GetParam(PARAM_TEXT);
 assert(tmp);
 memcpy(text,tmp+2,*(tmp+1));
 memset(text+(*(tmp+1)),0x00,1); // make the string

 GetParamDWord(COUNT,msgNumber);
 printf("Text received: %s\n",text);

 // If the message sequence number is less than given  limit,
 // continue message counting. If not stop the program.
 msgNumber++;

 // Prepare and send the message.
 // Change automata state to MESSAGE.
 PrepareNewMessage(0x00,MSG_MSG);
 AddParamDWord(COUNT,msgNumber);
 SetMsgToAutomata(FSM_TYPE_AUTOMATA);
 SetMsgToGroup(INVALID_08);
 SetMsgObjectNumberTo(idToMsg);
 sendToTCP();
 SetState(MESSAGE);
}

void NetAutomata::NetAutomata_MSG_MSG(){
 GetParamDWord(COUNT,msgNumber);
 msgNumber++;
 if(msgNumber < MAX_MSG_NUM){
  // Forward the message.
  PrepareNewMessage(0x00,MSG_MSG);
  AddParamDWord(COUNT,msgNumber);
  SetMsgToAutomata(FSM_TYPE_AUTOMATA);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(idToMsg);
  sendToTCP();
 }
 else {
  printf("Stop automata: %u\n",GetObjectId());
```

```
  // Prepare and send the message.
  // Change automata state to IDLE.
  PrepareNewMessage(0x00,MSG_STOP);
  SetMsgToAutomata(FSM_TYPE_AUTOMATA);
  SetMsgToGroup(INVALID_08);
  SetMsgObjectNumberTo(idToMsg);
  sendToTCP();
  SetState(IDLE);
 }
}

void NetAutomata::NetAutomata_MSG_STOP(){
 printf("Stop automata: %u\n",GetObjectId());
 SetState(IDLE);
}
void NetAutomata::NetAutomata_UNEXPECTED_IDLE(){
 printf("Unexpected message in the state IDLE \n");
}

void NetAutomata::NetAutomata_UNEXPECTED_MSG(){
 printf("Unexpected message in the state MESSAGE \n");
}

void NetAutomata::StartDemo(){
 uint8 *msg = GetBuffer(MSG_HEADER_LENGTH);
 SetMsgFromAutomata(FSM_TYPE_AUTOMATA,msg);
 SetMsgFromGroup(INVALID_08,msg);
 SetMsgObjectNumberFrom(0,msg);

 SetMsgToAutomata(FSM_TYPE_AUTOMATA,msg);
 SetMsgToGroup(INVALID_08,msg);
 SetMsgObjectNumberTo(0,msg);

 SetMsgInfoCoding(0,msg); // 0 = StandardMessage
 SetMsgCode(IDLE_START,msg);
 SetMsgInfoLength(0,msg);
 SendMessage(MBX_AUTOMATA_ID,msg);
}

uint16 NetAutomata::convertNetToFSMMessage(){
 // Manipulate only data because automata sends the new
 // message to itself.
 int length = receivedMessageLength-MSG_HEADER_LENGTH;
 memcpy(fsmMessageR,  protocolMessageR+MSG_HEADER_LENGTH, length);
 fsmMessageRLength=length; // mandatory - used by  workWhenReceive()

 // Rotate bytes
 uint16 msgCode =  GetUint16((uint8*)(protocolMessageR+MSG_CODE));

 switch((msgCode)){
  case IDLE_START:
   msgCode = IDLE_START;
   break;
  case IDLE_MSG:
   msgCode = IDLE_MSG;
   break;
  case MSG_MSG:
   msgCode = MSG_MSG;
   break;
  case MSG_STOP:
   msgCode = MSG_STOP;
   break;
  default:
   msgCode = 0xffff;
```

```
 }
 return msgCode;
}

void NetAutomata::convertFSMToNetMessage(){
 // Here we send the whole message.
   memcpy(protocolMessageS,fsmMessageS,fsmMessageSLength);
 sendMsgLength = fsmMessageSLength;
}

uint8 NetAutomata::getProtocolInfoCoding(){
 // Standard msg info coding
 return 0;
}
```

The file *NetAutomata.cpp* contains the definition of the class *NetAutomata*. This definition starts with the class constructor that first calls the base class constructor specifying no timers, two states, and the maximum of three state transitions per state for this automata type. After this, the constructor calls the function *SetDefaultFSMData*, which sets the data specific for this automata type.

The functions *GetMessageInterface*, *SetDefaultHeader*, *GetMbxId*, *GetAutomata*, *SetDefaultFSMData*, *NoFreeInstances*, and *Initialize* are the same as in the previous example. The only difference is that the name of the class *Automata* has been renamed to *NetAutomata*.

The function *NetAutomata_IDLE_START* handles the message *IDLE_START* in the state *IDLE*. First, it sets the message sequence number *msgNumber* to the value 1 and the identification of the destination automata instance *idToMsg* to the value 0. Next, this function prepares and sends the message, "THIS IS THE FIRST MESSAGE," to its peer in the remote FSM system by calling the function *SendToTCP*. At the end, it performs the state transition from *IDLE* to *MESSAGE* by calling the function *SetState* and specifying the value *MESSAGE* as its parameter.

The function *NetAutomata_IDLE_MSG* handles the message *IDLE_MSG* in the state *IDLE*. First, it prints the received message to the monitor. It then prepares and sends the message with the code *MSG_MSG* to its peer by calling the function *SendToTCP*, and sets the current state of this automata instance to the value *MESSAGE*.

The function *NetAutomata_MSG_MSG* handles the message *MSG_MSG* in the state *MESSAGE*. First, it gets the message sequence number from the received message and increments this value. It then checks if the new value of the message sequence number has reached the given limit. If not, this function prepares and sends the message *MSG_MSG* to its peer at the remote FSM system by calling the function *SendToTCP*. If it has reached the limit, this function prepares and sends the message *MSG_STOP* to its peer at the remote FSM system, and sets the current state of this automata instance to *IDLE*.

The function *NetAutomata_MSG_STOP* handles the message *MSG_STOP* in the state *MESSAGE*. It is fairly simple and just sets the current state of this automata instance to *IDLE*. The unexpected event handlers in this example just print the warning messages. In real-world applications, these

functions would trigger some higher-level recovery mechanisms. The function *StartDemo* creates the first message in the system. It fills in its header as if the automata instance with the identification 0 had sent that message to itself and sends this message to the mailbox assigned to this automata type.

The function *convertNetToFSMMessage* just copies the payload of the external message received from the remote FSM system to the current FSM system internal message (the last received message), because in this simple example, the two communicating instances have the same IDs and no need exists for any mappings between them. The pointer *fsmMessageR* points to the current internal message, the pointer *protocolMessageR* points to the current external message, and the variable *fsmMessageRLength* is equal to the payload size of the current external message. At the end, this function determines the message code and returns it as its return value.

The function *convertFSMToNetMessage* copies the whole new internal message to the new external message and sets the value of its length. The pointer *fsmMessageS* points to the new internal message, the pointer *protocolMessageS* points to the new external message, and the variables *fsmMessageSLength* and *sendMsgLength* contain their lengths.

The function *getProtocolInfoCoding* returns the code of the standard message coding (code 0x00) used for coding external messages. Note that in this simple example, both internal and external messages are actually standard messages.

File *Constants.h*:

```
// FSM
#define FSM_TYPE_AUTOMATA 0

// MBX
#define MBX_AUTOMATA_ID 0
#define MAX_MSG_NUM 10
#define COUNT 1
#define PARAM_TEXT 2
#define IP_ADDRESS "192.168.0.57"
#define PORT_1 7000
#define PORT_2 8000

enum AutomataStates {
 IDLE = 0,
 MESSAGE,
};

enum Messages {
 IDLE_START = 0,
 IDLE_MSG,
 MSG_MSG,
 MSG_STOP
};
```

The file *Constants.h* first defines general symbolic constants. It is very similar to the file with the same name in the previous example. The identification of this automata type *FSM_TYPE_AUTOMATA* is assigned the value 0, the identification of the mailbox related to this automata type

MBX_AUTOMATA_ID is assigned the value 0, the maximal message sequence number *MAX_MSG_NUM* is assigned the value 10, the identification of the message parameter that contains the message sequence number *COUNT* is assigned the value 1, and the identification of the message parameter that contains the text *PARAM_TEXT* is assigned the value 2.

The main difference with the previous example is the definition of the symbolic constants related to the communication over the TCP/IP infrastructure. The IP address *IP_ADDRESS* is assigned the value 192.168.0.57, the TCP port number for the first server *PORT_1* is assigned the value 7000, and the TCP port number for the second server *PORT_2* is assigned the value 8000. Next, the identifications of the individual states of this automata type, as well as possible message codes, are enumerated. This part of the file is the same as in the previous example.

File *Main.cpp*:

```
#include "conio.h"
#include "Kernel/fsmsystem.h"
#include "Kernel/LogFile.h"
#include "NetAutomata.h"

// If the following line is not commented out we get the code for the
// server listening to the port number PORT_1.
// If the following line is commented out we get the code for the
// server listening to the port number PORT_2.
#define AUTOMATA1

// Assume the following.
// The FSM system hosts a single automata type.
// The FSM system uses a single mailbox for the message exchange.
// Create the FSM system.
FSMSystemWithTCP fsmSystem(1,1);

// Create the instance of the class NetAutomata.
NetAutomata instance_1;

DWORD WINAPI ThreadFunction(void* dummy){
 uint32 buffersCount[3] = {5,3,2};
 uint32 buffersLength[3] = {128,256,512};
 uint8 buffClassNo = 3;

 // Initialize the FSM system.
 printf("Initialize the FSMSystemWithTCP... \n");
 fsmSystem.Add(&instance_1,FSM_TYPE_AUTOMATA,1,true);
   fsmSystem.InitKernel(buffClassNo,buffersCount,buffersLength,1);
 LogFile lf("log.log", "log.ini");
 LogAutomataNew::SetLogInterface(&lf);

 // Server in machine number 1 will listen to the port  number PORT_1.
 // Server in machine number 2 will listen to the port  number PORT_2.
 // It does not matter which instance will establish  the TCP
 // connection by calling the function  establishConection().
#ifdef AUTOMATA1
 printf("Start server...on port:%u\n",PORT_1);
 fsmSystem.InitTCPServer(PORT_1,FSM_TYPE_AUTOMATA);
#else
```

```
 printf("Start server...on port:%u\n",PORT_2);
 fsmSystem.InitTCPServer(PORT_2,FSM_TYPE_AUTOMATA);

#endif
 // Start the FSM system.
 printf("Start the FSM system...\n");
 try {
  fsmSystem.Start();
 }
 catch(...) {
  OutputDebugString("Exception - stop the FSM system...\n");
  return 0;
 }
 OutputDebugString("The end of the operation.\n");
 return 0;
}

void main(int argc,char* argv[]){
 DWORD threadID;
 bool end = false;
 char ret;

 // Start the FSM system thread.
 HANDLE hTemp = CreateThread(NULL,0,ThreadFunction,NULL,0,&threadID);
 Sleep(100);

 // Program works until the character 'Q' or 'q' is pressed.
 while((!end)) {
  if(_kbhit()) {
   ret = _getch();
   switch((ret)) {
    case 'Q':
    case 'q':
     fsmSystem.StopSystem();
     end = true;
     Sleep(100);
     break;
    case 'S':
    case 's':
     instance_1.StartDemo();
     break;
    case 'E':
    case 'e':
// Press 'e' to establish the connection with the remote server.
// This will enable the communication with the remote system.
#ifdef AUTOMATA1
     instance_1.setPort(PORT_2);
     instance_1.setIP((IP_ADDRESS));
     printf("establishConection on port:%u",PORT_2);
     instance_1.establishConnection();
#else
     instance_1.setPort(PORT_1);
     instance_1.setIP(IP_ADDRESS);
     printf("establishConection on port:%u",PORT_1);
     instance_1.establishConnection();
#endif
     default:
     break;
   }
  }
 }
 CloseHandle(hTemp);
 printf("The end. \n");
}
```

The file *Main.cpp* starts with a list of the necessary included files and the definition of the symbolic constant *AUTOMATA1*. This constant should be defined for the local process and not for the remote process (this is done by commenting out the source code line that defines the symbol *AUTOMATA1*).

Next, the instantiation of the class *FSMSystemWithTCP* is performed by a call to its constructor. The parameters used in this call specify that an instance of the *FSMSystemWithTCP*, named *fsmSystem*, will include a single automata type and this automata type will use a single mailbox. After that, a single instance of the class *NetAutomata* is made, *instance_1*. Additionally, this file contains the definitions of the FSM system thread function *ThreadFunction* and the function *main*.

The function *ThreadFunction* first prepares the data needed to define three buffer types. The sizes and quantities of these buffers are five at 128 bytes, three at 256 bytes, and two at 512 bytes. Next, the three automata instances are added to *fsmSystem*. Note that the fourth parameter of the first call to the function *Add* is set to the value *true*, which means that the instances are to be used as a pool of instances of the same type. After that, this function initializes the kernel by calling the function *InitKernel*, defines and sets the logging interface by calling the function *SetLogInterface*, starts the TCP server by calling the function *InitTCPServer*, and starts the *fsmSystem* by calling its function *Start*.

The function *main* starts the FSM system thread (which executes the function *ThreadFunction*) and suspends itself for 100 ms. After this, it waits for the user command. If the user presses the character 'E' or 'e', it establishes the TCP connection with the remote TCP server by calling the function *establishConnection*. If the user presses the character 'Q' or 'q', it terminates the program.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Index

D