DESIGN FOR TESTABILITY OF COMMUNICATION PROTOCOLS

By

ANTONIO ALFREDO FERREIRA LOUREIRO

B.Sc. (Computer Science) Federal University of Minas Gerais, Brazil, 1983

M.Sc. (Computer Science) Federal University of Minas Gerais, Brazil, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

December 1995

Department of Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date: 18 December 1995

Abstract

There is growing consensus that some design principles are needed to overcome the ever increasing complexity in verifying and testing software in order to build more reliable systems. Design for testability (DFT) is the process of applying techniques and methods during the *design* phase in order to reduce the effort and cost in testing its implementations. In this thesis, the problem of design for testability of communication protocols is studied.

A framework that provides a general treatment to the problem of designing communication protocols with testability in mind and some basic design principles are presented. Following the protocol engineering life cycle we have identified and discussed in detail issues related to design for testability in the analysis, design, implementation, and testing phases.

We discuss two important aspects that affect the testing of communication protocols: testing taking the environment into consideration and distributed testing. We present a novel algorithm and the corresponding design principles for tackling an important class of faults caused by an unreliable environment, namely coordination loss, that are very difficult to catch in the testing process. These design principles can be applied systematically in the design of self-stabilizing protocols. We show that conformance relations that are environment independent are not adequate to deal with errors caused by the environment such as coordination loss. A more realistic conformance relation based on external behavior as well as a "more testable" relation for environments which exhibit coordination loss are introduced.

We also present a novel algorithm and the corresponding design principles for checking dynamic unstable properties during the testing process. The method proposed can be used in distributed testing of communication protocols and distributed programs in general. This technique can also be used in normal execution of the protocol implementation to tackle

the problems of state build-up and exception handling when a fault is detected. A specific type of communication protocol, namely 3-way handshake protocols, is used to show it is possible to check general properties using this algorithm.

A comprehensive survey of testability and design for testability in the software domain is also included in the thesis.

Contents

List of Tables

List of Figures

Acknowledgement

It would be impossible to properly acknowledge everyone who helped me before and during my Ph.D. program at UBC. They made it a most rewarding period, both academically and otherwise.

First of all I am very grateful to the Department of Computer Science at the Federal University of Minas Gerais, Brazil for allowing me to pursue my graduate studies at UBC. Without the help, support, and incentive of Professors José N.C. Árabe, Alberto H.F. Laender, Geraldo R. Mateus, José M.S. Nogueira, Clarindo I.P.S. e Pádua, Nivio Ziviani, and other faculty members I would never have come here.

I am deeply indebted to my supervisors Prof. Samuel T. Chanson and Prof. Son T. Vuong for their professional support throughout my graduate studies. They taught me about research and helped me to understand the process of accomplishing it.

I am very grateful to the other members of my thesis committee: Prof. Norm Hutchinson and Prof. Jeff Joyce, the other two members of my supervisory committee; Prof. Cyril Leung and Nicholas Pippenger, my university examiners; and Prof. Rachida Dssouli, my external examiner. Special thanks must go to Dr. Pippenger for accepting being my university examiner at the last minute (literally!).

The University of British Columbia has an outstanding Computer Science Department. I would like to thank everyone there for their help and support. Special thanks and acknowledgement must go to the faculty members Norm Hutchinson and Jack Snoeyink, and Dr. Jeff Joyce for sharing their experience as researchers and leading research meetings in their areas of expertise; the staff members Gale Arndt, Peter Cahoon, Carlin Chao, Moyra Ditchfield, Sunny Gulati, George Phillips, Peter Phillips, Joyce Poon, Mike Sanderson, Monika Silbermayr, Carol Whitehead, Deb Wilson, and Grace Wolkosky; the graduate students MyungChul Kim, Charles Mathieson, Roland Mechler, Sijian Zhang, and especially Marcelo Walter for taking care of so many things while I was away from UBC; the visiting researchers Carlos Goulart and family, Holger Janssen and Rolf Velthuys at UBC, and Jeff Tsang and Andreas Ulrich at HKUST (Hong Kong); and finally John Buchanann (Univ. of Alberta) and Prof. Osvaldo Carvalho (UFMG, Brazil) for carefully reading this thesis.

Undoubtedly, Adriana Guimarães Loureiro (Ph.T.[1]—Put Hubby Through) deserves the lion's share of the thanks. I love you so much! This work and success is as much yours as mine.

This research was supported in part by CNPq, Brazil, and the Canadian Institute for Telecommunications Research during my stay in Canada, and FAPEMIG, Brazil, during my stay in Hong Kong.

[1]The first reference I saw to this term was made by Kenneth Barker, a faculty member at the University of Manitoba.

Chapter 1

Introduction

1.1 Motivation

Since the early history of computers, testing has played a central role in the development of software. Maurice Wilkes writes in his book entitled Memoirs of a Computer Pioneer his first insights about the development of software and the testing process [190, page 145]:

> By June 1949 people had begun to realize that it was not so easy to get a program right as had at one time appeared. (...) I was trying to get working my first non-trivial program, which was one for the numerical integration of Airy's differential equation. (... then I realized) that a good part of the remainder of my life was going to be spent in finding errors in my own programs. Turing had evidently realized this too, for he spoke at the conference on "checking a large routine".

Twenty years later, Wilkes' first impressions became evident to the computer science community when two terms were coined: software engineering and software crisis. Since the late 1960s, new principles, methods, methodologies, and tools have been proposed to improve the state of the art in software. Despite all efforts done in almost three decades, the problem of designing and implementing reliable software remains very serious, and as pointed out by Brooks [23] it seems that there is no *silver bullet*. Furthermore, incomplete, inconsistent and erroneous systems are delivered every day. Many of them with serious implications.

Software reliability is one of the fundamental problems in computer science. The two classical approaches that have been used to solve this problem are testing and verification.

Among these approaches, testing has been the method most used to guarantee the correctness of concrete computations. This is reflected in the estimate that, in general, "testing

consumes at least half of the labor expended to produce a working program" [10]. In practice, verification is used only in specific parts of a software. For instance, in the critical modules of a system.

There is a growing consensus that some design principles are needed to overcome the ever increasing complexity in verifying and testing software in order to build more reliable systems. These principles are expressed as verifiability and testability properties respectively.

Testability means that a piece of software has some properties, characteristics, or features that will facilitate the testing process. Design for testability (DFT) is the process of applying techniques and methods during the *design* phase in order to reduce the effort and cost in testing its implementations.[1] How easily the testing process can be realized depends on many factors. For instance, it depends on the goals of testing or the testing strategy (e.g., length of test sequences, testing time, and coverage); how the implementation under test (IUT) is viewed: black-box, grey-box or white-box, and the problem domain or application. To develop better DFT techniques it is very important to understand the two fundamental criteria: (i) the kind of application we are dealing with, and (ii) the testing process. These two criteria will be used throughout this thesis.

Testability is a property that should be introduced in the design for people who will test the product (software or hardware) and not, in general, for the end users. But depending on the application and/or requirements, some testing facilities can and should be made available to the end user.

1.2 Context of thesis

In this thesis we study the problem of design for testability of communication protocols. Informally, a communication protocol defines a set of rules and procedures that govern the communication among entities in a distributed system.

In practice, it is only possible to have entities communicating with one another if the definitions of the communication protocols that govern interactions among the systems are unambiguously defined. This is one of the major roles of a protocol specification: to

[1]See Section 2.2 for a comparison of testing and testability in hardware and software.

provide an unambiguous and precise definition of protocols. In order to avoid the ambiguity of natural languages, various formal description techniques (FDTs) have been proposed, defined, and standardized [112].

It is obvious that if a specification is not correct according to some requirements, its implementations will not be able to exhibit the behaviors intended by the designers. Therefore, a crucial point in the development of any communication protocol is the definition of a design specification that satisfies the requirements specification. Actually, this is a general rule applicable to other engineering applications as well. The analysis and design phases define a fundamental dichotomy found in software engineering and other areas of computer science: *what* to do (analysis) and *how* to do (design).

The design is performed with a clear goal: to meet all requirements identified in the analysis process. The ideal way to prove (or disprove) the logical correctness of a design specification of a communication protocol is to apply some *verification* methods to check whether some desirable properties hold (e.g., safety and liveness properties). This is a complex problem and it is not difficult to show that the verification of a simple property such as absence of deadlock (safety property) is PSPACE hard [144]. When verifying the correctness of communication protocols we have to be aware of these complexity bounds. This means that we cannot avoid the testing process where a protocol implementation must be checked for conformity to its specification. Testing is also a difficult problem since exhaustive testing of implementations of complex protocols (e.g., OSI protocols) is both theoretically and practically not feasible. Therefore, the goal of conformance testing is to detect as many errors as possible in an implementation using the minimum number of test cases. (See Section 3.4 for a definition of different types of protocol testing.) However, most communication protocols have been designed and implemented without conformance testing in mind.

In the protocol area, the term protocol engineering was coined to denote the protocol development cycle [112, 137]. This new area includes disciplines such as formal methods, software and knowledge-based engineering principles and basically follows the traditional software life cycle comprised of the following phases: requirements analysis and specification; design and specification; coding and module testing; integration and system testing; and

delivery and maintenance.

Following the protocol engineering life cycle, we have proposed a framework to reason about design for testability of communication protocols as discussed in Chapter 3. This framework will be used throughout this thesis.

Communication protocols are concurrent reactive systems and it is recognized that testing of concurrent programs is harder than testing of sequential deterministic ones as explained in the next section.

1.3 The problem of testing concurrent reactive systems

The problem of testing concurrent programs is in general harder than testing sequential deterministic ones. Some of the difficulties in testing concurrent programs are (i) the probe effect, (ii) non-reproducible behavior of the system, (iii) increased complexity, and (iv) lack of a synchronized global clock.

When the concurrent program is also reactive, another level of difficulty is added: (v) the state build-up due to its continuous interaction with its environment. These five factors are discussed in more detail in the following:

The probe effect. A probe is a name given to a piece of hardware and/or software introduced into a system to observe its internal behavior. The purpose of observation may be to gain better knowledge about the behavior of the system or to make sure that some constraints are not violated.

The probe effect is a name given to the fact that the behavior of a system may be changed when attempting to observe it. This is the main impediment to achieve observability of a system.

The kind of effect caused by the introduction of a probe into a system depends on how its monitoring is performed: intrusive or interfering. Basically, the probe may interfere with the relative timing among processes, prevent certain timing or coordination related errors from occurring, or may introduce faults that would not be present without it.

There are at least three different ways of addressing the probe effect [121]. It can be

ignored, i.e., it is assumed that the effect will seldom or never occur; minimized if the probe is implemented efficiently; or avoided by implementing the probe through logical clock (i.e., hiding it), or leaving the probe in the production system (it may be difficult to avoid), or using dedicated hardware to monitor the system.

Non-reproducible behavior of the system. It is very important to be able to reproduce the system's behavior for a given input when the system is being tested or debugged. Otherwise it will be more difficult to realize these tasks. This is the main assumption when performing regression testing in sequential software. In fact, achieving reproducibility in sequential deterministic programs is relatively trivial since the choice of control paths depends on inputs to the program and its initial state.

In concurrent software this is not the case. Its behavior is not reproducible because of nondeterminism either due to concurrency or introduced in the specification, or due to non-observability as explained in Section 3.1.1. In fact, without any mechanism to control a nondeterministic selection it is not possible to establish an upper bound on the number of attempts required to reproduce a particular computation.

The usual solution to testing of concurrent programs is to create a log file of event occurrences and replay the log file when debugging the system. This approach is not trivial, introduces the probe effect, and may be difficult to achieve due to lack of controllability.

Increased complexity. Complexity is increased because of greater difficulty in understanding, specifying, and testing the ways processes interact among themselves in a distributed application. In other words, it is very difficult to anticipate all possible scenarios that might occur when nondeterminism, communication, and synchronization are taken into account in a system that comprises of multiple execution threads. These threads occur concurrently in physically separated locations but may interact with one another from time to time.

Lack of a synchronized global clock. A distributed system can be viewed as a set of cooperating processes communicating with each other through message passing. In this situation it is generally not possible to determine precisely (i.e., deterministically) the total

order of events in the system because of the lack of a global clock. It is only possible to give a partial order. For example, a send event p must happen before the corresponding receive event in process q, and for some events there is no way to tell which one occurred first. This often leads to errors in specifications.

State build-up. A computation can be seen as a sequence of steps that determine the states reached from the initial state. In this sense, state build-up happens in any system although, this problem is intensified in concurrent reactive systems since local states are generated based on interactions among cooperating processes, thereby increasing the difficulty in understanding and specifying such systems.

As stated above, a reactive system interacts continuously with its environment. This problem is illustrated in the following. Consider a protocol implementation I where its local state can be defined informally as the values associated with its local variables and the contents of its communication channel at a given moment in time. The initial local State 0 is shown in Figure 1.1 with two circles. When a new event happens its local state changes. This is shown in Figure 1.1 by a transition labeled with $\langle e \rangle$. Usually, an event $\langle e \rangle$ is classified into one of the following three types: a send event, a receive event, or an internal event. A send event causes a message to be sent, and a receive event causes a message to be received. An internal event such as timeout or an interrupt is not related to the communication channel. We shall assume the environment in which I is embedded (e.g., an operating system or application) does not change the value of variables or the contents of the communication channels.

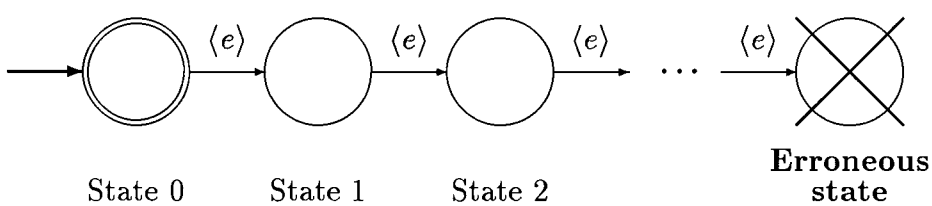


Figure 1.1: The problem of state build-up.

This example illustrates how valid input events $\langle e \rangle$ may lead to an erroneous state in the protocol implementation I. When the implementation I reaches an erroneous state it

may either continue to run but producing erroneous output or may crash and stop.

An erroneous state was reached when the values associated with its local variables and the contents of its communication channel are not consistent with the constraints or requirements given in the protocol specification. This means the global state of the system is not correct anymore once an erroneous state is reached. Furthermore, this problem is difficult to detect. Note that this behavior can only be identified if the test sequences are at least as long as the length of the faulty path. Even so, the test sequences may not detect this problem as described in Chapter 4.

In fact, Parnas, van Schouwen and Kwan [135] identify this situation as a critical problem to reactive safety-critical software. They suggest that this kind of software should be initialized periodically to avoid the state build-up problem.

1.4 Important points in the design process

The importance of systematic software design has been recognized since the term software engineering was coined in the late 1960s. Probably the most important reason to systematic design is related to the fact that the development of complex systems involves a large amount of detail. If the complexity is not kept under control it will be very difficult to achieve the desired results. The principles established by different design methods serve to *guide* designers through the complexity and detail where they might otherwise become lost.

A second and very important reason for systematic design is its impact on software quality. There are several characteristics that are desirable in a software system. One of the most important is reliability. All these properties are affected by design decisions.

A third reason for systematic design is to define a structure for the software system.

It seems to us that there are four fundamental points that should be observed when developing new design for testability techniques that are general enough to apply to any problem domain. They are described in the following:

1. **Definition of the testing goals to be accomplished.**

   Given a specific issue that affects the testing process, testing goals should be clearly defined since they will be used to develop the testability requirements to be introduced

in the design. The goals will guide the entire development process. Therefore, they should be expressed in terms of assumptions, requirements, constraints, and what is expected to be accomplished.

2. **Identification of the factors that affect the testing goals to be accomplished.**
   The development of a new DFT technique is a first step in this direction. Once the factors that affect the proposed testing goals are identified and clearly understood it is possible to start reasoning about them.

3. **Identification of the factors that can improve the testing goals to be accomplished.**
   This is probably the most difficult part, i.e., finding the solution/factors that should be incorporated into the design. The goal is to identify these testability requirements that will lead to an implementation easier to be tested according to the testing goals.

4. **Definition of the process to achieve the desired testing goals.**
   This is the development process. It should provide either an algorithm or a methodology to implement the solution found in the previous steps. This process involves a lot of creative work and a deep understanding of the entire protocol development process.

Ideally, the process of designing a new DFT technique should comprise *synthesis* and *analysis* techniques. The product of such process is a design specification that should lead to a correct and easily testable implementation. In a broader sense, synthesis denotes the process of building a software or hardware entity using primitive or predefined elements according to an algorithm. The final entity should have certain desired properties by construction. Analysis denotes the process of decomposing an entity into its elements and analyzing them using an algorithm. The goal of this process is to identify some undesirable properties or errors.

The synthesis and analysis processes should be based on a formal notation such as a formal description technique or a programming language. Otherwise it will not be possible

to define precisely or prove how or whether the goals of synthesis and analysis can be accomplished by applying certain rules and techniques.

In the context of design for testability, synthesis denotes the process of specifying a design specification that will meet some testability requirements according to some testing goals whereas analysis denotes the process of examining a design specification to check whether some testability requirements are met according to some testing goals. The applicability of each technique will depend on the issues under consideration.

Testability is, in general, a qualitative property but depending on the testing goals can be analyzed quantitatively. Whenever possible, testability metrics, which should consider both the control and data parts of the protocol, should be given.

It is very important to have software tools that help designers to meet testability requirements. In the hardware domain, commercial tools have been available for some time. In [32] a survey on tools used for protocol development based on formal description techniques (FDTs) is presented and none of them have facilities to incorporate testability into the design.

The next chapter presents a comprehensive survey of design for testability in the software domain that discusses all these aspects.

1.5 Main contributions

Design for testability of communication protocols is a broad and complex research subject. Furthermore, it is in continuous evolution since communication protocols with new requirements and constraints are being developed for new applications and environments. Of course, the new protocols need to be tested considering all of the new aspects.

Design for testability is not the kind of problem that once it is solved the solution is general enough to cover all its aspects. In fact, a testability requirement when introduced in the design will help specific testing goals. Chapter 2 will address this point in more detail.

Our contributions cover the following aspects of design for testability:

1. *A framework for reasoning about design for testability of communication protocols.*

   This framework provides a general treatment to the problem of designing communication

protocols with testability in mind. Following the protocol engineering life cycle we have identified and discussed in details issues related to design for testability in the analysis, design, implementation, and testing phases.

It is essential that the designer has a clear conceptual understanding of a complex activity in order to comprehend its complexity and understand the details. Without such a framework, the designer can only recognize individual facts and techniques whose interrelationships may not be clear. Furthermore, it is more difficult for the designer to understand a new problem in a global context without a framework.

These are the main reasons that motivated the framework proposed.

2. *A set of novel design principles to design self-stabilizing protocols and a new conformance relation.*

   Self-stabilization is one of the design principles for having well-formed protocols as explained in Section 3.2.2. If a protocol is self-stabilizing, then it is guaranteed that avoidance of an important class of faults known as coordination loss is incorporated into the protocol design. These errors are very difficult to catch in practice since they happen in unanticipated situations and it is difficult to incorporate this fault model into test case generation algorithms.

   We present a set of novel design principles for designing self-stabilizing protocols systematically. Two examples of real protocols and their self-stabilizing versions are given. Furthermore, we define a new conformance relation and a "more testable" relation for environments which exhibit coordination loss.

3. *A set of novel design principles to check dynamic unstable properties.*

   Informally dynamic properties define desired or undesired temporal evolutions of the behavior of a communication protocol. In protocol testing, most of the properties of interest that characterize valid or invalid behaviors are inherently unstable since there is no guarantee that they will remain either valid or invalid in a protocol computation. These are properties that are not amenable to be checked by traditional verification methods.

We present a set of novel design principles to check dynamic properties that will improve the testing process by proposing a mechanism to check global predicates defined in terms of local states and identifying places to check these properties. As we will see, this is a new approach compared to global monitors. This set of design principles also improve the reliability of a protocol implementation since it can be used to avoid the problem of state build-up as explained in Section 1.3, and can be used as a mechanism to introduce exception handling in the protocol implementation. Furthermore, these principles can be extended to build global monitors that check single points of global observation.

As can be seen, our contributions stress the *design for* concept that is the main goal of developing this piece of work. In Chapters 3 to 5, which form the main contributions of this thesis, we compare what we have done to other proposals presented in the literature to put our work in context.

1.6 Outline of the thesis

The thesis is divided into six chapters where this is the first one. Chapter 2 presents a comprehensive survey of testability and design for testability in the software domain. The goal is to present the current state of the art and show and discuss current methods that have been proposed in the literature. Chapter 3 presents a framework to study the problem of design for testability of communication protocols and is related to the first contribution mentioned above. Chapter 4 presents a set of novel design principles for designing self-stabilizing protocols and defines two new relations based on an environment that exhibits coordination loss. This is related to the second contribution. Chapter 5 presents a set of novel design principles to check dynamic unstable properties that will improve the testing process and the reliability of a protocol implementation. This is related to the third contribution described above. Chapter 6 presents our conclusions and suggestions for future work. Finally, Appendix A presents another example of a real protocol and its self-stabilizing versions.

Note that the framework presented in Chapter 3 together with the principles discussed in Section 1.4 define the core principles to be used in the design for testability of communication protocols. Chapter 4 discusses the "external" aspects that affect testing of communication

protocols whereas Chapter 5 discusses the "internal" aspects. In fact these two chapters have some fundamental common principles, namely the concepts of cycles and paths in protocols. The contributions to design for testability of communication protocols can be applied to other problem domains in distributed systems as well.

Chapter 2

Testability and design for testability in the software domain: A survey

In this chapter we give an extensive survey and discussion on testability and design for testability (DFT) in the software domain. We make a distinction between testability and DFT since some of the work reviewed here does not consider the problem from the design point of view. The goal is to present the current state of the art and discuss current methods that have been proposed in the literature. As we shall see, these methods suggest specific solutions to the different testing problems depending on the testing goals and problem domains. This fact is to be expected since there is no testability solution that covers all possible testing problems even for the same problem domain. In other words, there are different "testability views" even for the same testing problem.

This chapter is organized as follows. Section 2.1 presents other factors besides testability to integrate product and process design end to end. Thus testability can be seen as part of a broader design context. Section 2.2 gives some fundamental differences between testing and testability in the hardware and software domains. These differences are reflected in several aspects that influence the way hardware and software are tested and how testability is incorporated into the design process. Section 2.3 presents some definitions and properties of testability according to software engineering principles. Section 2.4 discusses different approaches to analyze and introduce testability in a piece of software following the traditional software life cycle. Section 2.5 looks at testability from the point of view of observability and controllability. These two issues are the basic requirements for hardware testability. Section 2.6 addresses testability in two specific application domains, namely communication

protocols and distributed architectures for real-time systems, following a discussion on the use of formal methods for DFT. It is important to emphasize that, in general, each work covers aspects of the other categories as well. The goal of this classification is to highlight the main points of each work so they can be put into perspective. Section 2.7 discusses four other approaches to software testing related to testability. The methods are randomly testable functions, program result checking, self-testing/correcting programs, and certification trail. Section 2.8 compares the methods presented in this chapter according to different criteria to get a global view on their relative strengths and weaknesses. Finally, Section 2.9 presents the conclusions for this chapter.

2.1 Testability in a broader context

The last decade saw many companies around the world making significant progress in the integration and improvement of the entire product realization process (PRP). The costs and time to realize a product were reduced, quality was improved, and more importantly, a reevaluation of the PRP was done in order to streamline the product process. In this context a new term was coined: design for X (DFX). DFX is an approach to integrate product and process design end to end. The goal is to design product and processes that will lead to cost-effective and high-quality operations in their entire life cycle. The X in DFX stands for manufacturability, testability, installability, compliance, reliability, safety, serviceability, and other downstream considerations beyond performance and functionality. The term DFX is a natural extension to other well-known terms such as DFT (design for testability), DFM (design for manufacturability), and DFA (design for assembly). This is similar to what happened in the hardware domain when the concept of design for testability was introduced in order to design ICs that are more testable and consequently manufacturable.

The important point here is that DFX preserves the *design for* concept, i.e., all downstream considerations are taken into account in the design phase. The interested reader is referred to [4, 82] for further information on DFX.

2.2 Testing and testability in hardware and software

Hardware designers have realized early that some design principles are needed to overcome the ever increasing complexity of testing integrated circuits (ICs). ICs should be designed to contain facilities to allow the circuit to be more easily testable and consequently manufacturable. Presently, testability methods are applied before or concurrently with system logic design and not as an afterthought when the circuit is completed. This is called design for testability. These methods propose circuit modifications or addition of special test circuitry on the IC to allow better test coverage.

2.2.1 Differences between hardware and software

In this section we present some differences between hardware and software that will help us to understand the problems faced by software designers. These differences are reflected in several aspects when testing hardware and software. The points discussed below are presented in no particular order.

Complexity. Probably the first difference that stands out between hardware and software is their complexity. The amount of documentation and time necessary to specify and thoroughly understand a simple software system is much more than it takes to specify and understand a simple hardware system. For example, the behavior description of a latch is in general easier to describe than the semantics of an assignment command in a programming language.

Tolerance. There is no useful interpretation of tolerance for software, i.e., small errors can have drastic consequences (e.g., a few years ago, a Venus probe was lost because of a punctuation error). In hardware, every design and manufacturing aspect can be characterized, in general, by a tolerance, i.e., a specified range of the right value which still produces a correct result.

Failures along the time. Hardware typically fails because of manufacturing failures and wear-out phenomena. Software may fail at any time, even after years of correct use, when an unexpected situation occurs which was not anticipated by the designers and overlooked

by the testers. This is due to specification errors.

Constraints and characteristics. In general, hardware designers constrain the set of primitives used in the design and follow some very restrictive rules on how to apply them. This leads to a design that is often regular, modular, and hierarchical. On the other hand, software designers do not have such preoccupation since software is much more *flexible*. Software is more flexible in the sense that it allows one to build, modify, and add more *logic* directly to the product relatively easily and in a practical way. Probably, this is the most significant characteristic that distinguishes software from hardware.

Reusability. In hardware, it is a common practice to reuse designs due to the characteristics discussed above. In software, reusability is not a common practice despite the fact that it could be an important mechanism for reducing production costs. A promising approach in this direction is the *object-oriented* paradigm that has reusability as one of its key aspects. But, much more work needs to be done before reusable software components become commonly used by software engineers in the same way hardware components are used by hardware engineers.

Formalisms used to write specifications. Hardware designers are often familiar with some formal (or semi-formal) method to write a hardware specification. For example, behavioral Verilog and VHDL models are widely used in the hardware community. In software, it is the other way around. Formal methods are seldom used to write specifications which makes it difficult to automate some tasks in the software life cycle such as verification, implementation, and testing.

Cost. In hardware, a design error may cause the commercial failure of a product in today's marketplace since it might take several months before the error can be fixed. In software, it is common to have alpha, beta, ... releases of a product (when the software is used in an experimental basis and possible errors are identified and corrected) before a *stable* version is obtained. In other words, the process of deriving a non-trivial software implementation from its design specification that will function adequately from the start (not for testing purposes) seems to be a dream.

2.2.2 Differences between testing and testability in the hardware and software domains

In the following, we present some major differences between testing and testability in the hardware and software domains.

Goal. The goal of hardware testing is to determine whether the circuits, boards, modules, or systems work properly. These elements can fail because of the manufacturing process or due to aging or some external factor (e.g., cosmic rays and temperature changes). Therefore, in general, hardware testing is not intended to determine the correctness of the design. That goal is accomplished by design verification.

On the other hand, software testing is used as a complementary and mutually supportive technique to verification.

Complexity. The hardware testing process involves the creation of a set of *test vectors*, application of these test vectors to the element being tested, and finally the comparison between the expected and obtained results. Test vectors are generated according to some fault model. Ideally, the test vector should cause the effect of faults within the element being tested to be reflected in an output since internal circuits are not accessible directly through probes. In software testing, it is easy to insert testing code in a module without changing its behavior if the module does not have a timing constraint.

Often large quantities of the same chip are manufactured. Therefore, it is necessary and justifiable to invest in test harness which is the process of testing a piece of hardware or software in isolation from the rest of the system [58]. Software, on the contrary, is not manufactured in large quantities and test harness will be used less frequently, and perhaps on variants of the original module. Ideally, software test harness should be easy to build and to change.

In general, software is similar to a sequential circuit in the sense that the next state depends on the current one and its input value. In hardware, a combinational circuit depends only on its inputs. Researchers in hardware have recognized that it is more difficult

to develop DFT techniques for sequential rather than combinational circuits.

Cost. Depending on the kind of application in which a circuit will be used and the fault model assumed, once a single fault is found, the testing process does not need to continue and the faulty element is discarded. In software testing we are interested in finding as many faults as possible (not just the first one). This will also be the case if a hardware element (for instance, a board) needs to be repaired.

Fault model. Integrated circuits are often tested according to a fault model. It means that test sets are generated according to the expected fault patterns. When injected into the circuit the test sets are expected to detect a defect if the behavior of the defective element follows the behavior assumed by the fault hypothesis. Of course, there is no guarantee that a defective circuit will actually exhibit the fault specified in the fault model. Often different fault models are adopted depending on the kind of circuit being manufactured. In hardware, design for testability means that the chosen fault models are considered when designing the circuits in order to make them testable and consequently manufacturable.

Software testing is often not based on a fault model because it is very difficult to predict how a piece of software can fail. Instead, test cases are generated based either on the specification (ideally by an automated procedure) or on the implementation. For a description of the different methods used in software testing see for instance [10, 45, 130].

Standards. One of the main methods in designing an IC which takes testability into account is the built-in test (BIT) [83] that adds standard test circuits to the chip being designed and manufactured. The main reason for developing this method comes from the fact that the density of logic gates on integrated circuits has increased about an order of magnitude in the last 10 years while the number of pins remained about the same. Without these capabilities, it is very difficult to have adequate observability and controllability on the circuit under test. The ANSI/IEEE Standard 1149.1 [2, 117] defines a boundary-scan architecture for BIT functions that can be extended to provide complex, automatically initiated test suites called built-in-self test (BIST). The standard 1149.1 has been rapidly adopted by IC designers and manufactures since its approval in 1990.

In the software domain there are no such standards. Some concrete proposals have been made to improve testability only recently.

Tools. It is very important to have software tools that help the designer to meet testability requirements. In the hardware domain such tools have been available since the last decade, e.g., SCOAP (Sandia Controllability Observability Analysis Program) [67], COMET, and TEA (Test Engineer's Assistant) [75]. This is to be expected since DFT in the hardware domain has been an active research area for the last two decades.

DFT in software is a relatively new research area and the fundamentals are being established now. Therefore, there are only a few tools available that support testability in the software domain as we shall see in this chapter. In [32] a survey on tools used for protocol development based on formal description techniques (FDTs) is presented and none of them have facilities to incorporate testability into the design.

The hardware testing process as described above is summarized in Figure 2.1.

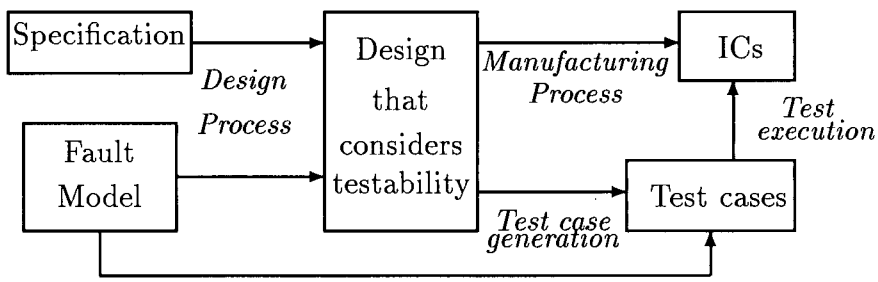


Figure 2.1: Partial view of the hardware testing process.

In hardware testing, test vectors are generated according to the fault model used to design the integrated circuit. Note that the specification is supposed to be correct according to some desired properties. Furthermore, it is supposed that the specification and/or design are presented in some formalism.

The interested reader is referred to [62, 83, 120, 168, 191] for further information on design for testability in the hardware domain.

2.3 Definitions and properties of testability

In the following we present some definitions of testability and design for testability that have been published in the literature in the context of software engineering. In Sections 2.4 to 2.7 other definitions will be presented when discussing specific methods related to testability and DFT. Each work is discussed in chronological order.

2.3.1 Boehm et al.

Boehm et al. [17] identify testability as one of the eleven important characteristics that software should have. The other ten are understandability, completeness, conciseness, portability, consistency, maintainability, usability, reliability, structuredness, and efficiency. They also present the relationships among the characteristics. For example, maintainability depends on understandability and testability. In other words, a maintainable program must be understandable and testable, or a high degree of maintainability suggests a high degree of understandability and testability. This is the only relation where testability is involved.

According to the authors, testable software facilitates the establishment of acceptance criteria and supports its evaluation. Furthermore, testable software has four properties:

- *accessibility*—facilitates the selective use of its components.
- *communicativeness*—facilitates the specification of inputs and provides outputs whose form and content are useful and easy to assimilate.
- *structuredness*—has a definite pattern of organization of its interdependent parts.
- *self-descriptiveness*—contains enough information for someone to determine its objectives, assumptions, constraints, inputs, outputs, components, and status.

2.3.2 Boehm

Boehm [18] identifies testability as one of the four major software requirements for design verification and validation criteria. The other three are completeness, consistency, and feasibility. A testable specification must be specific, unambiguous, and quantitative wherever possible. It means that generic goals, desired objectives, and non-functional requirements

should not be part of a specification since they are not precise. The idea behind testability here is to eliminate all vagueness and ambiguity of the specification in order to make it testable since requirements that are vague and ambiguous will have to be eventually tested. In [17] Boehm et al. show that early application of completeness and consistency checking leads to significant improvements in software error detection and correction.

The two books by Frakes et al. [59], and Martin and McClure [116] follow the same concepts presented by Boehm in [17, 18].

2.3.3 Beizer

In [10] Beizer presents two goals of testability: (i) to reduce the work of testing, and (ii) to have a testable code with fewer errors. How these goals can be accomplished depend mainly on the kind of application being developed. Independent of the application, Beizer suggests the use of a design style that will produce quality software. For instance, optimizations performed on the code can prevent this quality and should be done in very specific situations. On the other hand, a structured design based on the principles established in software engineering will be easier to test.

In his book, Beizer discusses flowgraph and path testing, transaction-flow testing, data-flow testing, domain testing, syntax testing, logic-based testing, transition testing, and metrics and complexity in testing. For each one of these topics it is given several testability tips that can be applied at different levels of abstraction. Each kind of testing is appropriate for a given application, or part of the software being tested, or phase of testing being performed. Therefore, as identified by Beizer, it is very important to the specifier, designer, implementer, and tester to have a clear understanding of what testing can and cannot do.

Beizer points out that the key to testability design is to build an explicit finite-state machine model and figure out how every part of it that has four or more states will be implemented. The idea is to have small finite-state machines with up to three states in such a way that it would be possible to do branch coverage in every possible state. This is reflected in the testability criteria presented for code where the goal is to keep the number of covering paths, the number of feasible paths, and the total path number as close to each other as possible.

2.3.4 Thayer and Thayer

Thayer and Thayer [167, page 667] define testability in the glossary of terms used in the field of system and software requirements engineering and supporting disciplines as follows:

1. "In software engineering, a software quality metric that can be used to measure those characteristics that provide for testing."

2. "In software engineering (requirements specifications), the extent that one can identify an economically feasible technique for determining whether the developed software will satisfy this specification. To be testable, specifications must be specific, unambiguous, and quantitative wherever possible." (Boehm [18])

2.3.5 Properties

Some properties of "easily testable" software have been identified in software engineering. Freedman [61] identifies four intuitive properties of a testable software:

1. Test sets are small and easily generated—it is not feasible or even possible to test a piece of software exhaustively.

2. Test sets are non-redundant—test inputs should not have repeated values. If the same test input yields different test results it might be an indication of a problem.

3. Test outputs are easily interpreted—before any test takes place it is necessary to write a test specification where test inputs and test outputs are provided. This will allow an effective and adequate test.

4. Software faults are easily locatable—ideally, software faults should be easily traced to specific inputs and/or parts of the software.

2.3.6 Some remarks

In [17] Boehm et al. present characteristics and properties of a testable software. In [18] Boehm emphasizes the points presented in [17] but discusses testability from the point of view of design (refer to the second definition of testability given in Section 2.3.4).

It is interesting to note that the properties of a testable software presented by Boehm et al. [17] can be associated to observability and controllability as discussed in Section 2.5. More precisely the definitions of the pairs communicativeness and self-descriptiveness, and accessibility and structuredness are related to observability and controllability respectively.

Most of the definitions, characteristics, and properties associated with testability are generic in the sense they provide an idea of what testability means and what can be accomplished since it is difficult to quantify it. This is reflected in the fact that there are just a few metrics related to testability (see Section 2.8). Beizer [10] is pragmatic in this aspect by providing several testability tips that can be applied at different levels of abstraction. He also recognizes the important point that testability depends on the kind of application being developed.

2.4 Testability according to the software life cycle

Different approaches have been proposed in the literature to analyze and introduce testability in a software product, and several of them will be discussed in this chapter. In this section, we present some approaches that address different testability aspects during the traditional software life cycle. Sections 2.5 to 2.7 discuss other aspects of testability and DFT.

2.4.1 Voas

Voas' work covers specification, design, coding, testing, and maintenance.

Voas defines testability as a prediction of the tendency for failures to be observed during random black-box testing of a program when faults are present. A failure will happen when the following conditions occur in the following order [129, 145]:

1. An input causes a fault to be executed.
2. After executing the fault, the program contains a data state error.
3. The data state error is propagated to an output state.

This chain of events is called the fault/failure model and relates program inputs, faults, data state errors, and failures. Therefore, a program with high testability will expose faults

triggered by input test cases whereas a program with low testability will prevent faults from being propagated to an output state.

Voas has proposed a static model and a dynamic model for predicting software testability [178].

The static model [176, 179, 180] can guide development during specification and design. The goal of the static model is the identification of program characteristics that make faults more difficult to find with random black-box testing and therefore will decrease software testability.

In the dynamic model [175, 177] attention is focussed on individual code locations during coding, testing, and maintenance. This model is closely related to the fault/failure model described above: analyze whether a fault will be executed during the testing process (the execution condition), analyze whether the fault will generate a data state error (the infection condition), and analyze whether the data state error will be propagated to an output variable (the propagation condition). This model assumes that faults occur at single locations. A location can be either an assignment, input statement, output statement, or a logical expression in a while or if statement.

2.4.2 Binder

Binder's work covers specification, design, implementation, testing, and maintenance.

Binder [12] defines testability as the relative ease and expense of revealing software faults. A testable implementation has the following properties:

- reduces the time and cost needed to meet reliability goals in a reliability-driven process, and
- provides increased reliability for a fixed testing budget in a resource-limited process.

Design for testability is defined as "a strategy to align the development process so that testing is maximally effective under either a reliability-driven or resource-limited regime."

Binder identifies six primary factors for improving the testability of an object-oriented system: (i) characteristics of the representation, (ii) characteristics of implementation, (iii) built-in test capabilities, (iv) the test suite, (v) the test support environment, and (vi) the

software process in which testing is conducted. He also discusses four possible strategies to introduce testability into the design of an object-oriented system: ad hoc DFT, structured DFT, standardized DFT, and self-test DFT. These strategies are similar to the ones used in the hardware domain [83].

2.4.3 Probert and Geldrez

Probert and Geldrez' work [142] cover specification, design and implementation. No definition for design for testability is given, rather they propose a grey-box testing technique called semantic instrumentation to enhance the software design for testability.

At the specification and design level, equivalence classes of behavior (ECB) are identified. The classes are used, for instance, to group different types of functional requirements present in the specification, decisions taken at the design level, and decisions to be taken at the implementation level. Each ECB receives a unique label. This could be seen as a kind of module decomposition.

The design is represented as a *design machine* (DM) that is an extension of finite state machines. A DM consists of states and transitions. States are design decision points and transitions are actions.

At the implementation level, a code segment is written for each equivalence class identified in the design. A one-to-many mapping function is established to map equivalence classes to code segments. A special code called semantic probe is added to each code segment corresponding to a class to detect whenever the code is executed and record this fact in terms of the corresponding design ECB.

The mapping is done to guarantee that activities performed during the development process such as design decisions and functional requirements are tested accordingly. This method can be used for documenting and verifying the completeness and consistency of the application.

In [142] Probert and Geldrez illustrate the method proposed by applying it to the service specification of the alternating bit protocol [9]. They show how to construct the design machine for the service specification, the mapping function, and the semantic coverage using semantic probes. These tasks are done manually and it would be interesting to see

how far they could be automated.

2.4.4 Bache and Müllerburg

Bache and Müllerburg [7] discuss testability when testing the control part of a sequential program. More specifically, they define testability measures for control flow testing strategies.

The authors define testability of a program as the effort needed to test it. The effort is given by the number of test cases (or test runs) needed for satisfying a test criterion. In particular, Bache and Müllerburg define testability for control flow based strategies as the number of paths or test cases required by a particular test strategy, and present some testability measures. In this case, the testability of a program will be the minimum number of test cases to provide total test coverage, if such coverage is possible. The testability measures are defined by metric functions, and as an example, they derive the metric values for branch testing.

It would be interesting to see these metrics applied to some examples, which it is not shown in their work. The metrics seem to be easy to be automated and the authors refer to a tool called QUALMS [7] that implements them.

Finally, as pointed out by the authors for complete testability of a program, it is necessary to develop testability measures for the data part as well.

2.4.5 White and Leung

White and Leung's work cover regression testing.

White and Leung [105] identify two types of maintenance: perfective and corrective. Perfective maintenance changes software functionality, and corrective maintenance corrects errors found in the software. In any case, after maintenance is done, the new version of the software must be tested again to check the correctness of its new and old functionalities. This is known as regression testing.

For small changes, some parts of the software may not be affected by the maintenance. In this case, regression testing needs to be executed at least on modules affected directly or indirectly by the maintenance. This can be accomplished by identifying a *firewall* around

the modules that need to be tested again. Ideally, in case a regression error is found in any of the retested modules, the firewall should not expand and include other system modules. Otherwise further regression testing has to be performed.

White and Leung [189] propose a more systematic approach to regression testing in software development based on the ideas described above called regression testability. They present an analysis method that identifies test cases to use as regression tests at the unit, integration, and system or functional testing levels that affect the modified parts of the software.

The method proposed by White and Leung is based on the assumption that errors, if present, are due to the modules modified during the maintenance phase. In practice this assumption may not hold true since it is not possible to rule out errors outside the firewall. However the idea of testing the modules within the firewall seems to be a sensible way of using testing resources.

White and Leung report their experience in applying these regression testing concepts in [106]. For small changes, the experiments showed considerable reductions in the number of required test cases.

2.4.6 Other methodologies

In this section we discuss two methodologies whose main goal is error prevention rather than error removal. Both methodologies span the entire software life cycle.

2.4.6.1 Cleanroom software engineering

The Cleanroom software engineering process covers specification, design, coding, testing, and maintenance [43, 125, 126]. It was proposed by IBM and is based on statistical quality control (SQC), a process introduced in manufacturing in the 1950s. Here, the measure of quality is given by the mean time to failure (MTTF) in a meaningful unit of time (real or processor time) of the developed software.

In the Cleanroom software engineering, SQC is applied to software development by separating the design process from the testing process. The three major activities in the method are (i) software specification that includes usage statistics, (ii) software development,

and (iii) software certification that is based on statistical testing. These activities define a cycle where the software is developed in an incremental way.

Mills et al. report in [126] their experience with three projects varying from 35,000 to 45,000 lines of code. In these cases, human verification was responsible for finding 90 percent of the total product defects before first execution. This was possible through a combination of formal design methods and mathematics-based verification which play a very important role in this methodology. Many programs were redesigned in order to apply simpler verification arguments and thus simplifies the correctness verification process. This technique can be viewed as design for verifiability.

2.4.6.2 Defect prevention process

The defect prevention process (DPP) covers specification, design, coding, testing, and maintenance [89, 118]. It was also proposed by IBM and was initially applied to the development of communications products such as VTAM, NCP, and NetView. This is a methodology that can be used in the development of different software products.

The DPP is based on the assumption that quality and productivity improvements in the development of software can be accomplished through (i) systematic causal analysis of errors, (ii) implementation of preventive actions, and (iii) feedback to developers. The main idea of this process is to use the actual errors found in a software to guide the development of future products. The defect prevention process relies solely on the actual defect data rather than on conjectures. Therefore, the testing phase plays an important role in this methodology.

This process defines a cycle that starts when errors are found during the testing phase of a software product, or in the production environment. The first step is to identify the causes of the error so preventive actions can be taken in order to avoid similar errors in the future. Eventually, the error, its causes, and the preventive actions taken are reported to the development team. Through this process, the developers can apply the experience gained to the development of new software and avoid making similar errors.

2.4.7 Some remarks

Voas [175, 176, 177, 178, 179, 180], Binder [12], Probert and Geldrez [142], Bache and Müllerburg [7], White and Leung [105, 106, 189], the Cleanroom software engineering process [43, 125, 126], and defect prevention process [89, 118] propose mechanisms to enhance the testing process according to the traditional software life cycle. Although these methods propose different approaches to improve the software quality, the methods proposed by Voas, Binder, the Cleanroom process, and DPP span the entire life cycle of a software.

Voas proposes a static model and a dynamical model for predicting software testability. Both models are based on metrics that indicate software testability. One of his conclusions is that there is a relationship between faults remaining undetected and the type of function containing the fault [176, 179, 180].

Binder identifies six primary factors that can improve testability of an object-oriented system and discusses four possible strategies to introduce testability into the design of an OO system. These strategies are analog to the ones used in the DFT of integrated circuits. Hoffman also took some of the ideas used in hardware testing and applied them in his work. In principle, the basic ideas proposed by Binder can be applied to non-object-oriented systems as well.

Binder advocates the necessity of having a test architecture and a test specification language as mechanisms to improve built-in test DFT. These ideas were formalized earlier by the International Organization for Standardization (ISO) that proposed a conformance testing methodology and framework [87] for testing communication protocols. The standard includes four different test architectures and a notation for specifying tests for communication systems [141].

The goal of the method proposed by Probert and Geldrez is to guarantee semantic coverage, i.e., that equivalence classes of behavior identified at the design level are tested accordingly. This differs from the usual structural testing strategies such as control flow strategies that try to guarantee some kind of coverage in the code, such as statement coverage where each statement is executed at least once.

The Cleanroom software engineering process and the defect prevention process aim at developing software with better quality by proposing new software engineering principles.

Basically these methods propose to pay more attention during the design process. The success of these methods depends critically on the knowledge and experience of the people who put them into practice when developing new products. Binder also identifies knowledge and proper education as key factors to achieving testability.

The Cleanroom process seems to work very well for large software products with relatively long histories since it is based on statistical quality control. On the other side, the defect prevention process depends on previous experience in developing other products. In both methods, testing plays a very important role through statistical testing and by collecting the actual defect data during testing.

Bache and Müllerburg measure testability for control flow testing by the number of paths or test cases required by a particular test strategy. Voas points out that metrics like this does not express the difficulty in testing a program with, for instance, a single control path since it may be very difficult to cover all the cases in a single path. This can be contrasted with the metrics proposed by Voas that are based on the semantics of the program.

White and Leung propose a systematic way of performing regression testing that can be used to save testing resources. This is also one of the principles proposed by Hoffman to do module testing as described in Section 2.5.1.

2.5 Observability and controllability

The concepts of observability and controllability (O&C) were first introduced in dynamical systems and automata [91]. Dynamical systems can be defined in terms of inputs, outputs, states, and state-transition functions. Observability and controllability have also been used for specifying hardware testability [120]. Recently, these concepts have been applied to software as well. In the following we present an overview of observability and controllability in the software domain. Each work is discussed in chronological order.

2.5.1 Hoffman

Hoffman [78] does not provide an explicit definition for testability but identifies observability and controllability as two important factors in design for testability. Observability is defined as the ease with which the required behavior of a module or submodule may be observed.

Controllability is defined as the ease with which an arbitrary stimulus may be applied to a module or a submodule.

Hoffman's work focuses on test case execution of individual modules. He proposes to do module testing based on three principles: (i) systematic module regression testing, (ii) design for testability based on the use of test scaffolding (a technique that uses test drivers and stubs to emulate the environment in which the module runs), and (iii) apply automation cost-effectively, i.e., automation should not be used on tasks that are cost-effective when executed manually or when it is unclear how they can be automated.

The motivation for developing the principles described above comes from the observation that general hardware testing concepts can be applied to the testing of software modules. Specific techniques of hardware testing are not very useful. The idea in software testing is to build more complex systems from simple, tested modules.

2.5.2 Dssouli and Fournier

Dssouli and Fournier [52] discuss observability and controllability in the context of communication software (communication protocols). The authors define testability as a property of the software that "includes testing facilities allowing the easy application of several testing methods, the detection of existing errors and their more rapid correction."

It is proposed that some testing facilities be introduced into the implementation to accomplish some testing goals, i.e., easy application of testing methods, detection of existing errors, and rapid correction. The testing facilities are abstracted by interactions points (IPs). The last part of the definition (detection of errors and rapid correction) can be defined as *debugging* which is the process of identifying the exact cause of an error and correcting it. The error detection and error correction are mainly based on an observable architectural structure of the implementation which should be obtained through refinement, decomposition, and introduction of interaction points.

Therefore, according to the definition it is possible to express testability as: Testability = Testing Facilities + Testing Goals.

Dssouli and Fournier suggest the introduction of a phase called *instrumentation and testability* before the implementation but after the design phase. In that phase, designers

can introduce internal interaction points into the implementation that will make its structure observable during the testing process. Consequently, all internal and external interaction points should be observable, so the tester can select a subset of them to be used during the testing phase.

At a given IP, an observer can observe interactions or traces between modules in the implementation which can be compared to the specification. Dssouli and Fournier distinguish the way an observer observes the traces and how the observer interacts with the modules being observed, i.e., the implementation under test (IUT).

The interactions can be observed either directly or indirectly. In the former case, an event is observed directly at a given IP, whereas in the latter case, the observer observes at IP_j events that are propagated from an unobservable IP.

During the observation of interactions, the observer can be either active or passive. An active observer can exchange messages with the IUT, and therefore, control the testing process. A passive observer just observes interactions.

Therefore, an active interaction defines a point of control and observation (PCO) since it affects the testing process. A passive interaction defines only a point of observation (PO).

Finally, an important issue is the granularity of the observation at the implementation level. Four levels of granularity are proposed:

1. Structural observation—for implementations obtained through modular decomposition. In this case it is possible to control and/or observe specified IPs.

2. Transition observation—for implementations based on transitions (automaton).

3. Input/output observation—based on the interactions between modules or between the IUT and the environment.

4. Functional observation—based on the instrumentation of the code.

The authors claim that the structural observation is the most appropriate for communication software and that it can be combined with the other three levels of granularity to enhance error detection and location.

Recently, Dssouli et al. [51] proposed a "model for testability transformation based on modification of a given specification. The testability of a protocol entity is assumed to be based on the shortest length of a test suite needed to achieve guaranteed coverage of certain faults."

2.5.3 Freedman

Freedman [61] proposes a new concept called *domain testability* which is defined in terms of two properties: observability and controllability.

- Observability—a software component is observable if distinct outputs are observed for distinct inputs.
- Controllability—a software component is controllable if, given any possible output value, an input exists that generates that value.

Domain testability refers to the ease of modifying a program so that it is observable and controllable.

Freedman points out that a software component is not easily tested when it has some input-output inconsistencies. Input inconsistencies mean that different test outputs are obtained for the same test input value. In this case, test inputs are incomplete and the outputs may be related to some other information (e.g., a given state) that is not known to the testers. If it is not the case, this may be an indication of an error. In general, deterministic programs do not have input inconsistencies. This kind of inconsistency is often present in a distributed system which is inherently nondeterministic, and in database management systems where their internal states often vary with time.

A software component has output inconsistencies when it is not possible to find a test input that will lead to a specific value in the domain of an output identifier.

Freedman defines domain testability as an extensional property of programs. If a program is domain testable it has no input-output inconsistency. As shown above, domain testability is defined in terms of observability and controllability. In order to have a program that is domain testable it is necessary to modify it. These modifications are called extensions and can be of two types: observable and controllable.

Observable extensions introduce program inputs based on implicit states, i.e., distinct outputs are observed for distinct inputs. Controllable extensions modify outputs.

Freedman presents new metrics to evaluate the effort required to modify a component so that it becomes domain testable. These metrics are based on the modifications that need to be introduced into a component in order to become observable and controllable. These extensions are proposed for two software components: expression procedures (similar to Ada function subprograms and other imperative languages that support the function concept) and command procedures (similar to Ada procedure subprograms and other imperative languages that support the procedure concept). The measures of observability and controllability for sequence, selection and iteration commands are also presented.

The ideas proposed by Freedman allow one to assess the testability of a functional specification by applying the concepts of observability and controllability. These measures will reveal whether there are hidden program states that need to be made explicit in the specification.

2.5.4 Saleh

Saleh [151] discusses observability and controllability in the context of communication software, more specifically in the service definition phase of a communication protocol.

Saleh presents two definitions related to testability:

- *design of testable protocols*—protocol designs whose implementations are easier to test, and
- *communication software design for testability*—the process of designing a software so that the testing of implementation becomes more economical and manageable.

Saleh reasons about testability during the service definition of a communication protocol. Testability requirements are treated as special-purpose communication services or functions which can be introduced into the specification of the protocol procedure rules using some construction mechanism such as synthesis or refinement. In his work, Saleh uses service-oriented protocol synthesis techniques [140] to synthesize testable procedure rules

from testable services. It is argued that it is simpler to specify testability requirements at the service level than at the protocol design level (procedure rules).

Saleh points out that testability requirements introduced into the original protocol design (procedure rules) make it more complex and increase the complexity and cost of its validation. On the other side, the complexity of a protocol derived from a testable service using synthesis techniques should not be of concern since the protocol is correct by construction.

It should be noted that the complexity of a protocol design is increased when testability requirements introduced into the design enhance its functionality in order to achieve some testing goal. If the testability requirements impose constraints on the design then the complexity of validation and testing may in fact decrease. Also, as pointed out by Probert and Saleh [140], the current state of the art in synthesis techniques has not yet reached the point where a complete protocol specification can be derived automatically from its service definition.

Saleh defines architectural and behavioral requirements that will enhance the observability and controllability of the communication services during the testing process. Observability is defined as the ease of observing the behavior of a particular protocol entity using both the upper and lower service access points (SAPs) as points of observation in the context of the OSI reference model [84, 195]. Saleh does not present a definition for controllability in communication software but we could define it in a similar way as the ease of controlling the behavior of a particular protocol entity using both the upper and lower SAPs as points of control

The architectural requirement is accomplished with the introduction of a special SAP called testability service access point or T-SAP. From the point of view of the OSI architecture, T-SAPs are similar to other SAPs but are only used for testing purposes. Furthermore, T-SAPs do not interfere with the communication service during the normal operation of the protocol.

Behavioral requirements are defined in terms of service primitives to be invoked exclusively at the T-SAPs in order to enhance observability and controllability. Also, these primitives are only used for testing purposes. The primitives related to observability are

Probe (P) to probe the state of a protocol entity, and Trace (T) to enable the trace collection of a given entity. The primitives related to controllability are Restart or Reset (R) to reset a given protocol entity to its initial state, and Set (S) to set a protocol entity to a particular state according to the variables or parameters specified in the service primitive.

However, the important point about these primitives is that they are low level and require expertise on the part of the user.

2.5.5 Kim and Chanson

Kim and Chanson [93] discuss observability and controllability in the context of formal specifications of communication protocols written in Estelle (see Section 3.2.3 for a brief description of Estelle).

The authors do not provide a definition of design for testability of communication protocols but rather discuss its goal which is to provide precise and efficient ways of testing the protocol implementation. This goal is achieved through a generic technique of instrumenting an implementation derived from a specification in Estelle.

Kim and Chanson identify three aspects in a formal specification written in Estelle that can be used to reason about design for testability. The aspects are data, transition and module. In the work, only the transition and module aspects are discussed.

In order to instrument these aspects in the implementation, the formal specification in Estelle is to be expressed in normal specification form (NSF). A specification in Estelle written in NSF consists of a single module without procedures and functions. The modular structure of the specification should be reflected in the implementation.

The proposed techniques are defined to instrument an implementation but can be used to enhance the testability of both sequential and concurrent specifications. In each case, the authors consider how controllability and observability can be accomplished.

Sequential specification.

- *Observability*—the proposed scheme introduces two primitives and the associated variables, transitions (to set/unset observability), and also adds an output statement with a transition id at the end of every conditional transition which allows the tester to ob-

serve exactly which transition has been executed. This defines a trace. The primitives are TEST_observe and TEST_unobserve and they work as a toggle switch enabling and disabling the observability.

As pointed out by the authors, if the implementation is instrumented for observability it might interfere with its normal operation.

- *Controllability*—the scheme introduces two primitives and the associated variables and transitions (to set/unset controllability) which allow to control the selection of a particular transition. The primitives are TEST_control and TEST_uncontrol and they work as a toggle switch enabling and disabling the controllability.

  The selection of a transition is given in the data field of the input interaction primitive Input_Primitive.data.

Concurrent specification. A concurrent specification is viewed as a set of single modules written in NFS. Again, each individual module in the specification is translated into an individual module in the implementation with the same structure.

- *Observability*—it associates the behavior of individual modules in the concurrent implementation to their related individual modules in the concurrent specification through the traces of individual modules.

  A concurrent module implementation conforms to the concurrent module specification iff the traces of the individual modules in the implementation are present in the individual modules in the specification. This defines an implementation relation that disallows any extension as defined by Brookes et al. [22].

- *Controllability*—it exposes internal interaction points that are not visible externally in the outermost module. It also defines a capability for dynamic configuration of the module structure that will allow to expose the internal interaction points and, consequently, the modules to be tested.

  Controllability on a concurrent specification allows to control the selection of specific modules to be tested. The controllability of a single module is achieved as described above.

2.5.6 Testing interfaces

Note that some of the methods discussed in this section make use of a testing interface to observe and control the implementation under test. Some of these methods also define primitives that can be used only through the testing interface.

Hoffman [78] suggests the introduction of a testing interface to be used for testing purposes only in order to improve the design for testability. No specific primitives are associated with the interface.

Dssouli and Fournier [52] propose the introduction of internal interaction points (IPs) in the instrumentation and testability phase so the implementation structure of a communication software can become observable during the testing process. In their case, IPs do not need to be used for testing purposes only. Furthermore, the IPs can be either passive or active. A passive testing interface is just an interface of observation whereas an active testing interface is an interface for control and observation. Dssouli and Fournier suggest the addition of new primitives as one of the alternatives to make an IP observable but no primitives in particular are defined.

Saleh [151] defines architectural and behavioral requirements in order to enhance the observability and controllability of a communication protocol. The architectural requirement is accomplished through a testing interface called testability service access (T-SAP). The behavioral requirement defines four primitives that can be used exclusively at the T-SAP. Saleh classifies the service primitives into passive and active. Both the T-SAP and the service primitives can be used only for testing purposes.

Kim and Chanson [93] define two pairs of service primitives related to observability and controllability respectively. The service primitives do not need to be sent through a testing interface, even though the trace is directed to a specific port created for observation.

In the context of object-oriented systems there are at least two methods that make use of a special interface. Binder [12] proposes the addition of a specific testing interface to all classes (see Section 2.4.2). Sloane [160] proposes the extension of an object definition to include specific interfaces that support events and extra methods for debugging purposes.

The simple existence of a testing interface and specific testing primitives do not guarantee the observability and controllability of the implementation under test (IUT). It is very

important that the structural design facilitates the observability and controllability of the IUT. Of course none of this can be done without the expertise of the tester. Therefore, the tester plays a central role in this approach.

Also note that observability and controllability are in general related to a testing interface and primitives. This idea is also present in design for testability in the hardware domain. In fact it is one of basic principles in testing integrated circuits.

2.5.7 Some remarks

Hoffman [78], Dssouli and Fournier [52], Freedman [61], Saleh [151], and Kim and Chanson [93] propose mechanisms to change or enhance the specification in order to produce an observable and controllable implementation.

Hoffman proposes four techniques to improve observability and controllability (O&C): use of information hiding, test scaffolding, test interface, and use of embedding executable assertions. Dssouli and Fournier suggest three alternatives to improve O&C (addition of primitives, selective broadcasting, and direct call to a trace procedure). Both Saleh, and Kim and Chanson suggest the introduction of service primitives to enhance observability and controllability of the protocol specification. This contrasts with the method proposed by Freedman where observability and controllability are achieved by changing the design itself.

Each work proposes a different approach to introduce observability and controllability into the design and/or implementation. Hoffman proposes some principles that should be followed. Dssouli and Fournier suggest the introduction of a phase called instrumentation and testability before the implementation and after the design where the internal structure of the implementation could become observable and controllable during the testing phase. Freedman uses an analysis technique to introduce O&C. In this case, the design is analyzed and eventually modified if there are input/output inconsistencies. Saleh uses a synthesis technique to introduce O&C where testable procedure rules are synthesized from testable services. Kim and Chanson use a simple instrumentation process to introduce O&C where the main assumption is that the modular structure of the specification is reflected in the implementation.

The ideas suggested by both Hoffman, and Dssouli and Fournier can be applied to a formal or informal specification. The method proposed by Freedman can be used to analyze functional requirements, design specification or low-level specification of a software component that can be expressed in an informal or formal way. Both Saleh, and Kim and Chanson suppose that the specification is written using a formal method.

Among these methods, the one devised by Freedman seems to be the most difficult to automate since it needs a great deal of expertise on the part of the designer. Also, the method proposed by Saleh is difficult to implement. The reason is that the current state of the art in synthesis techniques has not yet reached the point where a complete protocol specification can be derived automatically from its service definition [140]. The method proposed by Kim and Chanson can be implemented by a front-end or transformation tool or written concurrently with the specification and then fed to an Estelle compiler or interpreter. This method should not be difficult to implement in a different formal description technique such as Lotos and SDL. The degree of difficulty to implement the ideas proposed by Dssouli and Fournier depends on the alternative chosen. The principles proposed by Hoffman need to be further refined so they can be automated.

Another interesting point is how observability and controllability introduced into the system affect the testing process. Since the approach adopted by Freedman changes the design it does not interfere at all with the testing process. On the other hand, Hoffman, Dssouli and Fournier, Saleh, and Kim and Chanson perform what can be called an instrumentation of the implementation. This process is not completely passive and therefore can change the behavior of the testing process when compared to the same implementation that does not have any facilities to support observability and controllability. Both Saleh, and Kim and Chanson recognize this point. This is because there is some overhead associated with the observability and controllability facilities (e.g., to test whether the service primitive is one used for observability and controllability). Even if these facilities are left in the *production* version of the protocol, they may affect the behavior of the protocol because the overhead is still present, despite the fact they will not be used during normal operation.

Only Freedman proposes some metrics to evaluate the O&C introduced into the specification or implementation.

Besides the work discussed in this section, Voas [176, 179, 180] assumes observability in his description. He also proposes to increase observability by adding output parameters to critical modules. The goal is to "reveal as much of its internal state as is practical, since information in these states may reveal a fault that will otherwise be missed during testing."

It is interesting to note that an observable and controllable specification according to the method proposed by Freedman may have a high testability value according to the metric proposed by Voas. Yet related to observability Voas and Hoffman propose respectively the addition of self-tests and assertions into modules that are difficult to observe in order to check internal values during the computation. In [12], Binder presents these alternatives and others to improve observability and controllability of object-oriented systems (see Section 2.4.2).

Schütz [186] also discusses observability and controllability in the context of distributed architectures for real-time systems (see Section 2.6.2).

2.6 Application domain

In this section testability and design for testability are studied in the context of specific domains. This is another way of studying this problem and it is one of the fundamental factors that affect testability and DFT. The domains covered in this section are communication protocols, distributed architectures for real-time systems, and formal methods.

2.6.1 Communication protocols

Besides the work discussed in this section, Probert and Geldrez (Section 2.4.3), Dssouli and Fournier (Section 2.5.2), Saleh (Section 2.5.4), Kim and Chanson (Section 2.5.5), and Skou (Section 2.6.3.2) also discuss testability in communication protocols.

2.6.1.1 Yu

Yu [192] defines testability-directed specification as a specification expressed in a formal method that facilitates subsequent testing of its implementations. This definition is presented in the context of distributed systems in general and communication protocols in particular. Yu identifies two requirements that a formalism should satisfy in order to produce a testability-directed specification:

1. Strong expressive power—capacity of expressing concurrent behaviors that can be observed during the testing process.

2. Timing model that facilitates the testing process—a model that facilitates the generation of test cases.

Yu proposes an extension to Lamport's work on logical clocks [99] called relative clock time as a basis for specifying and testing distributed systems. This extension together with temporal assertions are applied to Parnas' trace assertion language (TAL) [134] yielding an extended trace assertion language (ETAL). TAL is a language used to specify a software module or system in terms of its external observable behavior. ETAL allows a system to specified in terms of relative time.

The relative clock time is the basis for the definition of the global events model that in turn is the basis for a new specification-based test result analysis method. This method can be applied to conformance testing of both service and protocol specifications. In particular, the global events model can be used to detect collisions in communication protocols. Collisions can be viewed as *cross overs* of interactions in a distributed system and may or may not cause testing problems.

The specification-based test result analysis method enhances testability by identifying spontaneous events and reconciling locally inconclusive test results resulting from collisions. In [192] Yu shows the applicability of this method to the specification of the ISO transport service in ETAL.

2.6.1.2 Drira et al.

Drira et al. [50] introduce the concept of refusal graph as the basis for testability degradation analysis in labeled transition systems (LTS). This analysis can be applied to formal description techniques such as Lotos.

Let I be an implementation derived from a specification S expressed in LTS that is embedded in an environment E. The testability of S is said to be good when the sets of rejected implementations obtained testing I through E and testing I directly (i.e., without E) are exactly the same.

Drira et al. proposed a method for analyzing the testability of I with respect to the verdict of the test execution when the implementation can only be tested through E. They showed two opposite cases. First, a verdict may erroneously validate a non-conforming implementation when tested through E but the verdict would discard I if the implementation were tested directly (degraded verdict). Second, a verdict may erroneously discard a conforming implementation when tested through E but the verdict would validate I if the implementation were tested directly (biased verdict). This is called testability degradation analysis [48, 49] and it is based on conformance testing.

Drira et al. [48] report a prototype tool developed in Prolog that implements the testability degradation analysis technique.

2.6.1.3 Petrenko, Dssouli, and Koenig

Petrenko, Dssouli, and Koenig [136] define testability as a property of software as proposed by Dssouli and Fournier (see Section 2.5.2). Furthermore, software testability implies in a software that "is easily testable, i.e., that the amount of work needed to identify the input and output data characteristics and the state of the implementation is small."

Petrenko, Dssouli, and Koenig propose a testability metric to compare different protocol implementations derived from the same protocol specification. The implementations are built as a composition of finite state machines (FSMs). The metric only considers the control part of the protocol specification that is modeled as an FSM.

When testing a protocol implementation it may be difficult to identify the best test suite with minimal length and maximal fault coverage. The test set may be infinite since a communication protocol is a reactive program and complete test coverage cannot be achieved unless a fault domain is defined a priori.

Based on this observation, the authors propose a testability metric for protocol implementations that is inversely proportional to the amount of testing effort. In this case the testing effort is considered to be the length of a test suite needed to achieve complete fault coverage assuming a pre-defined fault domain.

The testability metric proposed can also be used to determine where the points of observation should be located. The metric is directly proportional to the number of outputs.

Based on this fact, points of observation should be located in places that maximize the number of output combinations.

2.6.2 Distributed architectures for real-time systems

Schütz [186] compares two types of distributed architectures for real-time systems with respect to their testability, namely event-triggered and time-triggered systems. In an event-triggered system all actions are performed when an event happens. This type of system is characterized by the lack of a global clock. A time-triggered system performs their actions at given points in time. Therefore, this type of system requires a global clock and must execute time-triggered computations, protocols, and observations.

The testability of the two distributed architectures are compared using the criteria of observability, controllability, and test coverage.

Observability is the mechanism used for determining whether the system under test executes correctly or not. It should allow observation of events, and determination of correct ordering and timing of events. However, the behavior of the system may be changed when attempting to observe it. This is called the probe effect and is the main impediment to achieving observability.

Controllability is the mechanism used for controlling the execution of the system so its behavior can be reproduced. The presence of concurrent activities is one of the main reasons why system behavior is non-reproducible behavior. Some factors that influence the execution of theses activities are the processor load, the network traffic, nondeterminism in the communication protocol, the probe effect (if there is some kind of observation), and nondeterminism in the application. Of course, nondeterminism in the application affects the execution of both sequential and concurrent programs.

Test coverage indicates how many possible scenarios that can occur during any execution are covered by the test cases. Ideally, the coverage should be 100% but the potentially large number of event combinations limits the test coverage in practice.

In a time-triggered system, the time base is discrete and determined by the granularity of the action grid (minimum time interval between two consecutive actions). In this case, every input case can be *observed* and *reproduced* in the time domain for all the valid values.

This may not be possible in an event-triggered system. Therefore testing time-triggered systems is more systematic and more powerful.

In all three aspects, Schütz concludes that a time-triggered system is more testable than an event-triggered system. This is not surprising since time-triggered systems are based on a stronger assumption of having access to a global clock. Schütz also compares these aspects using a time-triggered system called MARS (MAintainable Real-time System) [97] and the same conclusion is reached.

It is important to note from this work that testability depends very much on the underlying system architecture. Therefore, the design of the system and the testing methodology used should take advantage of these properties.

2.6.3 Formal methods

A very important aspect that should be considered in the design process is the formal description technique (FDT) used to write the specification. FDTs are essential in every phase of the development process, i.e., the definition of validation methods, automatic or semi-automatic implementation, and test case generation. See Section 3.2 for a discussion of the design process and a brief introduction to the three FDTs — Estelle, Lotos, and SDL.

2.6.3.1 Testability as a criterion to compare formal methods

Larsen [103] presents four criteria for comparing specification formalisms, but only discusses the first two: expressivity, compositionality, decidability, and testability. Expressivity and compositionality are related to expressive power and abstraction level. Decidability is related to the feasibility of defining verification techniques; the more expressive a specification formalism is, the more difficult to define a simple and tractable mathematical model that can be used for verification purposes. Testability is related to the testing of a process or an implementation as discussed in this thesis.

Note that depending on the formalism used to write a specification some aspects may become very difficult to test. Some of these problems are discussed in Chapter 3.

2.6.3.2 Testability and probabilistic processes

Skou [159] introduces the concept of testable property, i.e., a property that can be validated with arbitrary high confidence provided it is possible to replicate the processes at any stage during the testing process and observe the parallel execution of the copies. The fundamental idea behind this notion is to assume that the non-deterministic choices of processes have a probabilistic nature and therefore give rise to a distribution function on the set of possible outcomes for a given experiment. In other words, the notion of probabilistic transition systems as a model for resolving nondeterminism is introduced. Probabilistic transition systems are like ordinary transition systems except that the transitions $p \xrightarrow{a} p'$ for a given pair (p, a) are assigned probabilities summing up to 1.

Based on this principle, Skou studies the following three property classes using the Hennessy-Milner logic [76] applied to finite processes: bisimulation equivalence classes [76], refinements within a modal process logic [104], and parameterized bisimulation equivalence classes [102]. These property classes are shown to be testable.

Using the definition of testable property Skou shows that for finite state processes it is possible to use testing as an approach to tackle the problem of correctness instead of using formal verification. In particular, the author showed how three class properties amenable to formal proofs can be confirmed or rejected with arbitrary high confidence by performing a test which is derivable from the property in question.

2.6.3.3 Testability and Lotos specifications

In [24], van de Burgt, Kroon and Peeters report their experience in specifying an image retrieval application in Lotos, and testing a module of it. According to their conclusions, testability can be improved by keeping the interface of the implementation as close as possible to the specification and limiting nondeterminism in the specification as much as possible. They also report some particular issues encountered when deriving the implementation from the Lotos specification.

2.6.3.4 Testability and SDL specifications

Ellsberger and Kristoffersen [56] report their testing experience of a communication protocol specified in SDL. The authors identify aspects of SDL specifications that are difficult to test, namely asynchronous communication, the time model used, and nondeterminism. They also present suggestions for improving testability in these aspects.

Luo, Das, and von Bochmann [113] present a method for transforming SDL specifications with "save" constructs into equivalent finite state machines without the "save" constructs so that test cases based on FSMs can be generated. The main motivation for this transformation comes from the fact that test case generation methods for SDL specifications prohibit the use of "save" constructs.

2.6.4 Some remarks

One of the most important aspects in the area of design for testability is the kind of application we are dealing with and how it is specified. Among the different application domains, design for testability of communication protocol is currently an area of active research.

Note how the different proposals focus on the DFT problem. Probert and Geldrez [142] look at testability from the point of view of semantic coverage. Dssouli and Fournier [52], Saleh [151], and Kim and Chanson [93] look at observability and controllability. Skou [159] and Yu [192] study the implications of the formalism used on testability. In fact, testability is one of the criteria proposed by Larsen [103] for comparing formal methods. Drira et al. [48, 50, 49] study how the verdict of the test execution is affected when testing an implementation through an environment and directly. Petrenko, Dssouli, and Koenig [136] examine the difficulty of testing an implementation based on the length of a complete test suite.

The comparison done by Schütz [186] shows that the stronger the assumptions about the system (in this case distributed architectures for real-time systems) the greater the testability. This conclusion is also reached in a similar way by Drira et al. when it is shown that a faulty implementation may not be detected when tested through an environment instead of directly.

2.7 Other approaches to software testing related to testability

Some techniques have been proposed to increase the reliability of a software system. These are non main-stream techniques that are applicable to specific software classes. One of them is the method of n-version programming. This method suggests the development of n different versions of a software system by different teams [35]. The n versions are executed in parallel and their results are compared (voted) to determine which value to use. This method is based on the belief that half or more of the n versions will be always correct for any input although there is no evidence that this is true in practice [96].

There are at least four other methods that are similar to n-version programming. The first three discussed below exhibit probabilistic behavior.

2.7.1 Randomly testable functions

Lipton [110] proposes a method to test programs that compute a mathematical function. The method does not guarantee absolute correctness of the implementation but rather that the implementation is correct with a certain probability which can be as close to one as desired. This probability is not stated in the empirical sense but it is a rigorous mathematical statement. The method is based on a distribution-free theory of random testing and the program is treated as a black-box.

Let P be a program that implements a function $f(x)$. The method works as follows:

(i) Determine that P is correct on a substantial fraction of all possible inputs.

(ii) Construct a new program Q that calls P for a set of random points. The values of P for the random points are used by Q to return the value of $f(x)$.

Functions that can be computed using this method are called randomly testable functions. Lipton presented examples of randomly testable functions in number theory and basic linear algebra. But what functions are randomly testable and how they are characterized are still open questions.

2.7.2 Program result checking

Blum et al. [13, 14] propose a new model called program result checking. Program result checking is the process of writing a result checker program that is run together with the original program. The result checker does not verify whether or not the program is correct for all inputs. Rather, the goal of the checker program is to verify only the result of the original program for specific inputs, one at a time. But there is still the problem of verifying the correctness of the checker for the program. In practice, result checkers are simpler than any program for a class of problem. Often, simplicity (in this case, that of the result checker) is easy to identify but it is not quantifiable.

Several result checkers have been published in the literature, including graph isomorphism, matrix rank, quadratic residuocity, linear programming, maximum matching, and greatest common divisor [13, 14, 148].

2.7.3 Self-testing/correcting programs

Blum et al. [15] propose the notion of self-testing/correcting. This is an extension of the theory of program checkers that was introduced by Blum et al. [13, 14]. A result checker is used to verify whether the program P which computes function f is correct for a given input. Therefore, the result checker does not provide the correct answer in case P computes f erroneously.

If program P computes f correctly on a substantial fraction of all possible inputs from a given distribution, which is determined by the function, then P can be turned into a self-corrected program that is correct on every input with high probability. In order to make sure that P is correct for the given distribution, the notion of a self-tester program is introduced. The self-tester should not be based on any program that supposedly computes f correctly. A possible way to achieve this requirement is to use a result checker as the self-tester program.

Blum et al. define self-testers and self-correctors in terms of probabilistic programs. Several self-testing/correcting pairs, mainly for different numerical functions, have been proposed in the literature [15, 148].

2.7.4 Certification trail

Sullivan and Masson [163, 164] introduce the concept of certification trail. The goal of this method is to provide the same capability of 2-version programming while using less resources. This method also uses two implementations when solving a problem but they run sequentially, one after the other. After executing the two implementations, the results are compared and considered correct if they are the same.

The main idea is to modify the algorithm of the first implementation to output some data that will be used by the algorithm in the second implementation. This data is called certification trail. For each algorithm, the certification trail should be chosen such that (i) the second algorithm will execute faster and/or have a simpler structure, and (ii) will not introduce any data dependency between the two implementations. The first requirement is the motivation to use this method compared to 2-version programming, but implicitly it says that the second implementation is different from the first one. The second requirement is needed so that the first implementation will not lead the second one to an erroneous result.

In [165], Sullivan et al. presented several certification trail examples for classic problems in computer science.

2.7.5 Some remarks

Several functions have been expressed in terms of randomly testable functions, and there are classes of problems that have result checkers, self-testers and self-correctors. However, no work exists that characterizes which functions and classes of problems can be expressed using these methods.

It seems that the first three methods above are suitable for testing programs that compute a mathematical function or for classes of problems where the output depends only on the input. These three methods cannot be applied directly as proposed for classes of programs whose output depend not only on the input but also on some internal state. Randomly testable functions cannot be defined since a function f may have a different behavior for each input. Result checkers cannot base their results or a specific input since P can give different outputs for the same input. Self-test is of no interest since the program can change

and become faulty as a consequence. Self-correcting cannot be achieved as well since there is no notion of having a program that has been certified to be usually correct.

Randomly testable functions, result checkers, and self-testing/correcting programs have probabilistic behavior. It is interesting to contrast these approaches to the Cleanroom engineering method that is based on statistical testing which takes into account both the usage pattern and the importance of the software components (see Section 2.4.6.1).

Certification trail and program result checker have at least two points in common: both check the result for a specific input and are supposed to be simpler and/or faster than the original program. Once a program result checker for a class of problem is found it is guaranteed that it can be used for checking any program in that class, independent of the algorithm used in the program. This is not necessarily true for certification since the certification trail used by the second implementation depends on the algorithm used in the first implementation. The second difference is that a program result checker is probabilistic and a certification trail is deterministic.

2.8 Comparison of methods

In this section the work surveyed is compared and classified according to various criteria. The goal is to put into perspective all methods discussed in Sections 2.4 to 2.7.

2.8.1 Testability or design for testability

Design for testability considers one or more aspects of the testing process during the design phase. The goal is to come up with a design that will lead to an implementation that is easy to be tested according to some testing goals. Testability analysis is performed during implementation or testing and has different goals such as assess the ease of testing the code, and suggest methods to generate, select, or evaluate test cases.

Table 2.1 shows the phases covered by each method according to the traditional software life cycle.

Work (Section)	DFT/T	Phases in the software life cycle				
		Spec.	Design	Impl.	Test.	Maint.
Life Cycle						
Voas (2.4.1)	DFT	•	•	•	•	•
Binder (2.4.2)	DFT	•	•	•	•	•
Probert (2.4.3)	DFT	•	•	•		
Bache (2.4.4)	DFT				•	
White (2.4.5)	T				•	
Cleanroom (2.4.6.1)	DFT	•	•	•	•	•
DPP (2.4.6.2)	DFT	•	•	•	•	•
O & C						
Hoffman (2.5.1)	DFT		•	•	•	
Dssouli (2.5.2)	T			•		
Freedman (2.5.3)	DFT	•	•	•		
Saleh (2.5.4)	DFT	•	•			
Kim (2.5.5)	DFT	•	•			
Applications						
Skou (2.6.3.2)	Check testability properties at the specification level.					
Yu (2.6.1.1)	DFT	•			•	
Drira (2.6.1.2)	DFT	•		•	•	
Petrenko (2.6.1.3)	DFT		•	•	•	
Schütz (2.6.2)	Comparison of two types of distributed architectures for real-time systems with respect to their testability.					
Other approaches	Randomly testable functions (2.7.1), Program result checking (2.7.2), Self-testing/correcting programs (2.7.3), and Certification trail (2.7.4) are related to testability in the sense that they try to improve the reliability of the testing process in a non-traditional way.					

Table 2.1: Testability and DFT according to the traditional software life cycle.

2.8.2 Factors that affect and improve testability

There are many factors that affect the testing process. For instance, the goals of testing or the testing strategy (e.g., length of test sequences, testing time, and coverage); how the implementation under test (IUT) is viewed (i.e., black-box, grey-box or white-box), and the problem domain or application. On the other side, different factors can improve testability depending on the kind of application and the testing process.

Table 2.2 shows the factors that affect (**A**) and improve (**I**) testability. Sometimes a given work does not suggest improvements to testability but goals to be accomplished. If the work does not discuss factors that affect or improve testability they are not presented in the table.

2.8.3 Other criteria

Synthesis or analysis. Ideally, the process of designing a piece of software should comprise *synthesis* and *analysis* techniques. The product of such process is the design specification that should lead to a correct and easily testable implementation. In a broader sense, synthesis denotes the process of building an entity using primitive or predefined elements according to an algorithm. The final entity should have certain desired properties by construction. Analysis denotes the process of decomposing an entity in its elements and analyzing them using an algorithm. The goal of this process is to identify some undesirable properties or errors.

The synthesis and analysis processes should be based on a formal notation such as a formal description technique or a programming language. Otherwise it will not be possible to define precisely or prove how or whether the goals of synthesis and analysis can be accomplished.

In the context of design for testability, synthesis denotes the process of specifying a design specification that will meet some testability requirements according to some testing goals whereas analysis denotes the process of examining a design specification to check whether some testability requirements are met according to some testing goals. The applicability of each technique will depend on the issues under consideration.

The second column in Table 2.3 shows whether the method applies synthesis (**S**) or analysis (**A**) techniques (**Tech.**). If the work uses both techniques analysis is always performed before synthesis.

The effect of introducing observability and controllability into the system. When introducing observability and controllability into the system, the system behavior may be changed. Therefore these facilities should be used judiciously otherwise incorrect conclusions

Work	Factors that affect and improve testability	
Life Cycle		
Voas	**A**:	information loss.
	I:	the metric DRR and the techniques PIE (Propagation, Infection and Execution) and propagation analysis.
Binder	**I**:	six factors: representation, implementation, built-in test, test suite, test tools, and process capability.
Probert	**I**:	semantic instrumentation.
Bache	**A**:	number of test cases needed for satisfying a test criterion.
	I:	testability measures based on coverage for control flow testing strategies.
White	**A**:	the maintenance may not affect all modules in a software.
	I:	identifying a firewall around modules that need to be tested again.
Cleanroom	**I**:	statistics of software usage, software development based on formal methods and mathematics-based verification, and statistical testing.
DPP	**I**:	systematic causal analysis of errors, implementation of preventive actions, and feedback to developers.
O & C		
Hoffman	**I**:	systematic module regression testing, test scaffolding, and automation.
Dssouli	**I**:	observable architecture of the software.
Freedman	**A**:	input and output inconsistencies.
	I:	domain testability.
Saleh	**A**:	testability requirements introduced at the protocol design.
	I:	architectural and behavioral requirements.
Kim	**I**:	data, transition, and module aspects in a formal specification.
Applications		
Skou	**I**:	testable properties.
Yu	**I**:	a formal method that has a strong expressive power and a timing model that facilitates the testing process.
Drira	**A**:	environment where the implementation is embedded.
Petrenko	**I**:	testability metric based on the length of a complete test suite.
Schütz	**A**:	the kind of application.
Other approaches	**I**:	making one or more calls when testing a program P.

Table 2.2: Factors that affect and improve testability.

may be reached.

The third column in Table 2.3 (**O&C**) shows whether observability and controllability

change the normal behavior of the system after introduction of these facilities.

Metrics for testability. Some of the methods described in this chapter provide ways of estimating testability according to a given criterion or assessing the effort needed to change the system to make it more testable.

The fourth column in Table 2.3 (**Metrics**) shows whether some testability metric is defined (Y for yes, and N for no).

Availability of a tool. Ideally, the method should be automated so the testability process can be performed faster and more reliably.

The fifth column in Table 2.3 (**Tool**) indicates whether there is a tool available that implements the method. Note that this information is based on the material cited in each work.

Applicability. Some methods seem more appropriate to test implementations of functions where the output depends only on the input values. This is similar to a combinational circuit where the output depends only on the inputs. Other methods consider implementations that have internal states similar to a sequential circuit. Furthermore, some methods consider only sequential programs while others accept concurrent programs as well.

The sixth column in Table 2.3 (**SW**) indicates the type of software that is suitable for each method. The following legend is used:

- **func**—indicates programs that implement mathematical functions. These programs do not have internal states.
- **seq**—indicates sequential deterministic programs.
- **any**—indicates any kind of program.

Application domain where the method has been applied. Some methods have been applied to a specific problem domain, though this does not mean that they cannot be applied to other domains.

The seventh column in Table 2.3 (**Appl.**) indicates the domain where the method has been applied. The following legend is used:

- **gen**—software in general.
- **OO**—object-oriented software.
- **CP**—communication protocols.
- **PP**—probabilistic process.

Types of methods. Basically, there are three types of methods: deterministic, randomized, and statistical.

The eighth and last column in Table 2.3 (**D/R/S**) indicates the type each method belongs. The following legend is used:

- **D**—deterministic.
- **R**—randomized or probabilistic.
- **S**—statistical.

2.8.4 Is DFT = observability and controllability?

There is a tendency to associate design for testability with observability and controllability. Among the work discussed in this chapter, Hoffman, Dssouli and Fournier, Freedman, Saleh, and Kim and Chanson look at testability considering observability and controllability. Schütz, Voas, Petrenko et al. and Binder also consider observability and controllability in their methods.

Probably this fact comes originally from the hardware domain where DFT capabilities are developed to provide adequate observability and controllability of the circuit under test. The main reason to have an observable and controllable circuit is to make sure that intermediate values are being computed correctly and are not being cancelled along the

Work	Tech. S	Tech. A	O & C Y/N	Metrics Y/N	Tool Y/N	SW	Appl.	D/R/S
Life Cycle								
Voas	•	•	N	Y	Y	seq	gen	D, R
Binder	•		Y	Y	N	any	OO	D
Probert	•		N	N	N	any	CP	D
Bache		•	NA	Y	N	seq	gen	D
White		•	NA	N	N	seq	gen	D
Cleanroom		•	NA	N	NA	any	gen	S
DPP	•	•	NA	N	NA	any	CP	D
O & C								
Hoffman		•	Y	N	N	any	gen	D
Dssouli	•		Y	N	N	any	CP	D
Freedman	•	•	N	Y	N	seq	gen	D
Saleh	•		Y	N	N	any	CP	D
Kim	•		Y	N	N	any	CP	D
Applications								
Skou		•	NA	N	N	any	PP	R
Yu	•		NA	N	N	any	CP	D
Drira		•	NA	N	Y	any	CP	D
Petrenko		•	N	Y	N	any	CP	D
Schütz	NA		NA	NA	NA	NA	RT	D
Other approaches								
Lipton	NA		NA	NA	N	seq	func	R
Blum (PRC)[1]	NA		NA	NA	N	seq	func	D
Blum (STCP)[2]	NA		NA	NA	N	seq	func	R
Sullivan	NA		NA	NA	N	seq	gen	D

NA means not applicable.

Table 2.3: Other criteria for comparing the methods.

output path. In general, intermediate values might be wrong not because of an error in the specification or design, but introduced by the manufacturing process.

[1]Program result checking (Section 2.7.2).
[2]Self-testing/correcting programs (Section 2.7.3).

Observability and controllability are also introduced into the software design with the same goal: to have an observable and controllable implementation. This goal is very important and its realization often requires changes in the design and/or implementation. But note that this is just one of the testing goals. In communication protocols, there are different testing goals as discussed extensively in Chapter 3.

In summary, design for testability is much more than observability and controllability (see, for instance, the remarks presented in Section 2.6.4). This fact will also be clear from the remaining chapters of this work.

2.9 Conclusions

After surveying different aspects and proposals for enhancing testability in the software domain, a question arises naturally: is there a general technique that can be applied in the design process so that the implementation derived from the specification will be more testable? The answer is definitely no. There are so many variables and conditions that need to be considered during the testing process that no single technique could cover all of them. This becomes clear from the framework proposed in Chapter 3 and the discussion in Section 2.8.4. This fact is also valid in the hardware domain where different testability techniques exist. Furthermore, as mentioned earlier, design for testability is in continuous evolution.

We can group testability methods into at least the following classes:

1. Methods that assess the ease with which inputs can be selected to achieve some structural testing criteria.

2. Methods that predict the probability that existing faults will be revealed during testing given a specific criterion for generating inputs.

3. Methods that propose a design strategy that will lead to an implementation which is easier to be tested according to some testing goals and application domain.

The first class includes methods that work at the implementation level. This contrasts to methods in the third class that work at the design level. The second class is intermediate

and includes methods that work at both the implementation and design levels. The first two classes define what can be called software testability analysis, and the third one is generally known as design for testability analysis.

Chapter 3

Design for testability of communication protocols: A framework

In this chapter we propose a framework to reason about design for testability of communication protocols. This framework provides a general treatment to the problem of designing communication protocols with testability in mind.

It is essential that the designer has a clear conceptual understanding of a complex activity in order to comprehend its complexity and understand the details. Without such a framework, the designer can only recognize individual facts and techniques whose interrelationships may not be clear. Furthermore, it is more difficult for the designer to understand a new problem in a global context without a framework.

Following the protocol engineering life cycle, we have identified issues related to design for testability in the analysis, design, implementation, and testing phases. In order to solve the problem of DFT it is essential to examine these distinct phases in detail. Each one of these phases will be discussed in detail in Sections 3.1 to 3.4. We present the conclusions for this chapter in Section 3.5.

Preliminary versions of this chapter appeared in [31, 182].

3.1 Analysis

In this section we present and discuss the factors that affect testing and testability in communication protocols. In each of the following two subsections, we present the reasons

that support our arguments and discuss how nondeterminism and coordination affect testing, and how testability can be improved. Note that the five factors that affect the testing of concurrent software as discussed in Section 1.3 are present here.

3.1.1 Nondeterminism

The following types of nondeterminism can be identified in communication protocols involved in distributed systems: nondeterminism due to concurrency, nondeterminism introduced in the specification, and nondeterminism due to non-observability.

3.1.1.1 Nondeterminism due to concurrency

A distributed system can be viewed as a set of cooperating processes communicating with each other through message passing. In general each process runs sequentially. In this situation it is generally not possible to determine precisely, i.e., deterministically, the total order of events in the system because of the lack of a physically synchronized global clock in most distributed systems. Generally, it is only possible to give a partial ordering of events. For example, we know that a send event in process P must happen before the corresponding receive event in process Q. However, for some other events, there is no way of telling which one occurred first.

This type of nondeterminism is inherent and inevitable in distributed systems.

3.1.1.2 Nondeterminism introduced in the specification

Nondeterminism was first introduced in a computational model by Rabin and Scott in their seminal paper on finite state machines (FSMs) [143]. They noted that requiring all FSMs to be described using deterministic transition functions would lead to cumbersome descriptions for even elementary operations.

Nondeterminism in communication protocols means that when a process is in a specific state and receives a particular input, it may respond in two or more different but valid ways. Which response will be offered is unpredictable (at the specification level).

Actually, in automata theory, a nondeterministic finite automaton (NFA) allows zero, one or more transitions from a state on the same input symbol. It is also possible to extend the model of NFA to include spontaneous transitions for the input symbol ϵ (empty symbol).

Often, protocol specifications that exhibit nondeterminism do not consider the null event ϵ, and have two or more transitions for the same input event. In other words, there is a nondeterministic choice and the selection criterion is not defined.

Of course, nondeterminism can be avoided at the specification level. However, there are at least three reasons to introduce nondeterminism in a specification: conciseness, performance, and flexibility. These considerations are discussed below. Actually, performance and flexibility are closely related because it is possible to improve the performance if (somehow) the protocol can be customized, i.e., if there is some degree of flexibility. Therefore, performance depends on flexibility but flexibility can facilitate other aspects (besides performance) such as operating cost. Performance and flexibility are both dependent on the environment in which the protocol is executed. By environment we mean the network (and its parameters such as speed, and reliability), software and hardware, and applications.

Conciseness. When specifying a protocol, often there are more than one valid event that can occur at a given moment (state). For each one of these events it is also common to have more than one valid action that can be performed. If nondeterminism is introduced to model the different valid behaviors, we do not need to assign a different state for each distinct input/output pair (i.e., event/action). This reduces the number of states in the specification and facilitates the understandability and readability of the protocol. Actually, it is possible to convert an NFA into an equivalent deterministic finite automaton (DFA), but the number of states of the DFA can be exponential in the number of states of the NFA.

Performance. This is related to the kind of environment in which the protocol will run. For example, suppose there is a sliding window protocol that can be used in both a local area network–LAN ($\sim 10^{6-9}$ bps) and a long haul network–LHN ($\sim 10^{3-6}$ bps) and the protocol does not provide selective repeat, i.e., the protocol does not retransmit a specific packet if some error occurs. For the LHN which typically has higher error rate than the LAN, a smaller window size should be chosen to minimize the number of packets that have to be retransmitted due to errors. On the other hand, a larger window size would be selected for the LAN environment to increase parallelism.

Another performance issue is related to timing. It is not desirable to make timing decisions at the specification level. For instance, the timeout value to retransmit a packet not yet confirmed depends, among other factors, on the round-trip delay which can vary in different environments.

Flexibility. Often the environment in which the protocol will be executed is not known in the design phase. Indeed, the same protocol might be implemented in several completely different environments. Thus, some decisions cannot and should not be cast in stone in the specification.

3.1.1.3 Nondeterminism due to non-observability

The observable behavior of a deterministic module whose interface is not directly accessible (e.g., because it is embedded in another module) will necessarily be described in a nondeterministic way. Even if the interface is directly accessible, it may not always be possible to observe what happens inside the module and that will give rise to nondeterminism due to non-observability.

Nondeterminism due to non-observability is also difficult to avoid since it is common to hide implementation details behind some standard interfaces (e.g., the OSI reference model).

3.1.1.4 Implications to testing

In the environment we are considering, protocol entities communicate only through message passing with some coordination mechanisms. In general, to design communication protocols that work properly under normal circumstances in this environment is not a difficult task. The implicit characteristics of concurrency in protocols have led to other problems such as livelock, deadlock, and nondeterministic behavior. The most challenging of these is nondeterminism which makes the other problems more difficult to deal with and reduces the effectiveness of traditional methods for program development.

Nondeterministic behavior means that two executions of the same system may produce different, but nevertheless, valid sequences of events. This can be contrasted with the execution of a deterministic sequential program where it is always possible to reproduce the same input/output sequence. In general, to test a sequential program, a set of input test

cases are specified and applied to an implementation, and the test results are compared to the expected behavior. If an error is detected, the program is run again (i.e., deterministic execution) with the same input test to collect some debugging information to help in correcting the program. Once the error is fixed, the testing process is repeated to check if the error has been corrected and that no new errors have been introduced.

In communication software (or concurrent software in general), it is much more difficult to exercise a particular sequence of events that will lead to an error since there is an enormous number of such sequences. It is also more difficult to locate the origin of an error from a given result because of nondeterminism. Even when the error is located and corrected, it is more difficult to run the system again using exactly the same steps that led to the error (again due to non-reproducibility). Furthermore, it is also more difficult to guarantee that no new errors were introduced.

Another problem is test case generation from specifications containing nondeterminism. Only recently has some work been done in test generation and fault diagnosis for protocols modeled as nondeterministic finite state machines [3, 63, 65, 95, 114, 94], whereas many results are already available for deterministic ones [158].

3.1.1.5 Improvements in testability

In practice it seems most protocols are implemented deterministically since deterministic behavior is desirable. If there are multiple choices, the implementer usually picks one according to some criteria (e.g., knowledge of the environment in which the protocol will be run). Given a particular testing goal it would be interesting to develop methods to predict whether a particular choice will eventually lead to that goal.

When there are multiple alternatives to the same input event, the designer must make sure that different choices are compatible among themselves. This is to guarantee that different implementations will be able to communicate with each other. Again, it would be desirable to devise mechanisms to show whether or not different choices are compatible and how they affect the testing process.

Finally, it would also be very helpful for the protocol designer to have some guidelines as to where and why (including the three reasons above) nondeterminism should be introduced

in (or removed from) the specification for different types of protocols.

3.1.2 Coordination

Coordination specifies the ways processes in a distributed system interact with one another through message exchanges (communication) and how those processes arrange their computation steps to accomplish some desired result (synchronization). Therefore coordination is realized by two different parts: communication and synchronization.

Communication is the spatial part since it defines the ways processes can exchange information. Synchronization is the temporal part since it defines the mechanisms used to order the computation steps among processes in time. Depending on how the synchronization mechanism works, processes can synchronize their computation steps explicitly by sending/receiving messages, or implicitly by ordering the messages in time.

Communication and synchronization together determine the correct way processes should coordinate. But sometimes the distinction between them it is not clear since some forms of communication such as remote procedure call (RPC) and Ada rendezvous require synchronization. These protocols explicitly synchronize the processes participating in the data transfer.

The next two subsections describe some forms of communication and synchronization used to coordinate processes.

3.1.2.1 Communication

In the type of distributed systems we are dealing with, cooperating processes do not share memory. Therefore communication via shared memory is not possible and interprocess communication is realized by exchanging messages. The following three forms (or paradigms) of interprocess communication can be used for defining communication primitives: message passing, remote procedure call (RPC), and transactions. These mechanisms are often discussed in detail in books on distributed operating systems and database management systems (see for example [68]).

In this work we are interested in interprocess communication based on message passing since this is the mechanism most often used in communication protocols. We assume that

messages are sent and received by invoking the communication primitives SEND and RECEIVE respectively. The result of executing a RECEIVE depends on whether and how many messages are available. If more than one message is available, it is necessary to determine which one will be received. This can be done by either defining some criteria or just picking one message at random.

A non-blocking SEND primitive, which allows the sender to continue its computation once the request to send the message is made, is called *asynchronous* communication, whereas a blocking SEND primitive is called synchronous communication.

3.1.2.2 Synchronization

The order of message passing in distributed systems is important as it may affect correctness or is required by the application to restrict access to a common resource. Cooperating processes must be synchronized or the result of computation could be wrong. The set of rules and mechanisms that define and implement such control is called *synchronization*. The control mechanisms are often provided by the underlying operating system. Furthermore, most distributed systems use asynchronous communication because it is difficult to synchronize clocks on the different processors. This makes synchronization more difficult to implement in distributed systems.

Synchronization problems arise because the cooperating processes do not have the same view of the global state since processes are autonomous and messages can have arbitrary network delays. Reachability analysis [187, 193] is often used to detect such problems. The problem with this approach is the so called *state explosion*, i.e., rapid growth of the number of global states as the number and complexity of each process and the underlying services increase. Several techniques have been proposed [80] to reduce the state explosion problem but it is still the main drawback of reachability analysis. In fact, for some complex protocols, the amount of reachable states is unbounded [188].

When a synchronization problem is found, the protocol must be redesigned. In that case the new protocol can become more complicated and difficult to understand, and new problems may be introduced. This leads to a continuous cycle of analysis and modification until an error-free protocol is obtained.

Another problem that can affect synchronization is coordination loss which can be caused by (among other factors) inconsistent initialization, transmission errors, process failure and recovery, and memory crash. Coordination is lost when the local states of each process, which may be correct individually, form an inconsistent global state. A self-stabilizing communication protocol (discussed below and in Chapter 4) overcomes the problem of coordination loss.

3.1.2.3 Implications to testing

The synchronization problem is relatively simple to solve if nodes and channels that participate in the computation do not fail. On the other hand, this problem can become intractable or even impossible when nodes and channels can fail. In other words, the difficulty in solving a synchronization problem depends on the fault model assumed [72]. However, as pointed out in Section 2.2.2, software testing is often not based on a fault model—this makes the identification of this type of error more difficult.

Among the synchronization problems, coordination loss probably happens most frequently in practice [71, 80, 100, 150, 162]. This problem is very difficult to be identified during the testing process since it occurs only in particular situations which are time dependent and may not be reproducible. This class of errors should be approached during the design phase.

3.1.2.4 Improvements in testability

An alternative approach to reachability analysis is to design protocols that correct coordination loss automatically. This technique is called self-synchronization. Miller [123], and Lin and Stovall [107] proposed techniques to build protocols that are self-synchronizing based on communicating finite state machines (CFSMs). Gouda and Mutari [71] proposed a proof technique using FSMs to prove the self-stabilization of a protocol. A self-stabilizing protocol can start the execution in any unsafe state (one that does not preserve an invariant) and is guaranteed to converge to a safe state after executing a finite number of state transitions without outside intervention.

Note that self-stabilization algorithms are synchronization algorithms but self-

stabilization is a property of the algorithm. This makes self-stabilization a paradigm for designing synchronization algorithms. For instance, asynchronous transmission using start and stop bits is self-stabilizing independent of how the receiver starts its reception. If the reception starts incorrectly there will be a framing error. Eventually the receiver will identify one or more stop bits and will re-synchronize (of course assuming that the hardware is working properly). It is also easy to show that the alternating bit protocol (ABP) can be self-stabilizing.

3.2 Design

The importance of the design process was discussed in Section 1.4. Note that testability is an issue of the design process. Furthermore, at the design level it is more difficult to separate testability issues from other issues as discussed in Section 6.2.

In this section, we discuss several issues related to protocol design. Specific issues that affect the implementation and testing phases will be presented in Sections 3.3 and 3.4 respectively.

3.2.1 Elements in protocol design

Protocol design plays an important role in many areas such as database, real-time systems, operating systems, computer architectures, computer networks and data communication. In practice, protocol design is not studied as a separate discipline but as part of a specific context which deals with issues related to a particular application domain.

In general, it is not a difficult task to design protocols that work properly under normal circumstances. It is the enormous number of possible sequences of events that may lead to errors that makes correct designs challenging to arrive at.

Thus, when designing a new protocol, care should be taken to identify and select the basic elements of a protocol specification[1] [80]:

1. *service*—what the protocol provides.

[1]In fact, these five elements are also present in protocols for different problem domains, other than communications.

2. *assumptions*—the conditions and environment in which the protocol will be executed.

3. *vocabulary*—the types of messages used to specify the protocol.

4. *encoding*—the format of each message.

5. *procedure rules*—the way two or more entities communicate by exchanging messages in order to perform some function.

The service and procedure rules (often called protocol specification) represent two levels of abstraction. The service specification describes the service offered by the layer in terms of valid sequences of interactions taking place at the upper boundary, i.e., at the interface. The protocol specification describes the logical implementation of a service in terms of the entities' behavior at a given layer. Therefore, the protocol specification can be viewed as a refinement of the service specification, and is often the most difficult part to design and verify, since it contains the rules responsible for accomplishing the goals of the protocols. In fact, there is a tendency to put emphasis only on this aspect.

In the following we discuss how each one of the five elements is related to testing and testability.

Service. The service specification defines what the protocol can provide to a user. It comprises the definition of service primitives, their parameters, and their temporal ordering. How the service can/will be realized depends on the assumptions made about the environment in which the protocol is to run. A service is provided through a service access point (SAP). A SAP can be viewed as a function with a well-defined interface. Different functions should be provided by different services (or primitives) since it will be easier to design, verify, implement, and test. This implies a modular design. Besides the SAPs provided to the user, special SAPs should be introduced for in-house testing purposes as discussed in Section 2.5.6. The idea of a testing interface is to provide access to the internal structure of a protocol entity during the testing phase. The designer should identify which internal aspects are important to testing. The next step is to build an implementation with testing interfaces preferably using synthesis and analysis techniques.

Some protocols offer different services and/or choices (due to nondeterminism) to the user. It is important to guarantee that two or more users using different services/choices will be able to communicate with each other.

Environment and assumptions. Traditionally, protocols have been designed taking into consideration the characteristics of the environment where they are going to run. This is despite the fact that when protocols are initially designed their dynamics in real networks are often not well understood. A possible solution is to simulate the protocol and its environment to have a better understanding of its behavior.

A problem frequently found in real networks is congestion, that is contention for insufficient resources. In the case of the TCP protocol, the approach for avoiding or recovering from congestion is to use delayed feedback mechanisms that exhibit complex behavior. Jacobson [88] observed that when TCP implementations attempted to send data through a congested network, the congestion was mostly due to a small number of connections sending data in the same direction via a single shared bottleneck link. Based on this observation, Jacobson proposed new congestion avoidance and control algorithms that are used in most current TCP implementations. Note that this problem was not anticipated at the design stage but only after observing TCP implementations in a real environment.

Another example is the application of higher layer protocols over satellite links. Chitre and Lee [36] observed that most of these protocols suffered performance degradation in this medium. Based on this fact, they proposed minor modifications to four higher layer protocols, namely class 4 transport protocol, session layer protocol, FTAM–file transfer, access, and management protocol, and network management protocol, to alleviate the performance degradation of these protocols operating in this environment.

For purposes of protocol design, a wide area network (WAN) differs from a local area network (LAN) in at least four ways: higher latency, lower bandwidth, point to point connections, and a higher probability of partitioning.[2] Furthermore, protocol entities located in different local area networks interconnected through a WAN are typically loosely coupled, i.e., their interactions are infrequent and asynchronous. The differences in characteristics

[2]The future ATM-based high speed networks will change this except for tail propagation delay which will not improve with transmission speed.

between WANs and LANs have a strong impact on protocol design for WANs, especially in protocols with multi-participants such as multi-phase commit or reliable multicast.

Another aspect in protocol design for WANs is that the physical medium has always been considered unreliable. Therefore, network protocols for wide area networks assume that packets can be duplicated, lost, damaged, or delayed arbitrarily. Given this environment, hosts can drop packets when they run out of buffers or other resources. For the sending entity, this situation will appear as a problem that occurred due to the unreliability of the network.

However, current networking technologies are offering more and more reliable services. It is expected that the communication environment in the next few years will allow the transmission of voice, image, and data in a simple, reliable, secured, and cost-effective manner [101].

Clark and Tennenhouse [38] propose new design principles to structure a new generation of protocols for future networks. In their approach, whenever possible, the different protocol functions are placed side by side, not on top of one another. In this context, layering is seen as a design principle used for semantic isolation of functional modules.

For some logical aspects of the protocol, the environment may deliver an invalid message to the protocol at a given state, or a valid message but not at the right state. The normal action taken by the protocol in these situations is to discard the message.

In summary, testing without sufficient knowledge of the environment may not be conclusive. Some assumptions about the environment should be made so the test case generation procedure can take them into account.

Vocabulary and Encoding. The vocabulary and encoding define the syntax of the protocol. Depending on the function of the protocol it is necessary to have some mechanism to detect transmission errors. The kind of application will dictate whether it is necessary to have an error-detecting (and correcting) code that works with high probability.

Procedure rules. Both nondeterminism and synchronization influence testability. Therefore the items discussed in Section 3.1 should be considered when designing the procedure

rules. Furthermore, the procedure rules should be realized according to the design principles described below. Otherwise, it will be very difficult to test some properties such as safety and liveness in a protocol implementation. The better approach is to verify those properties in order to avoid undesirable behaviors.

Note that if a formal specification S has a design error such as a deadlock, a correct implementation I of S should pass a conformance test if contains exactly the same error. In practice, an expert tester may be able to identify this problem when the test result is analyzed and it should lead to a design review despite the fact the verdict of the test is 'pass'.

On the other hand, a conformance test should fail if the implementation differs from the specification even if the specification contains errors. Here, we will rely on the protocol validation process to detect design errors.

Depending on the formalism used, some properties should also be considered when designing the procedure rules (see for example Sections 3.3 and 3.4).

3.2.2 Design principles

There are several design principles that should be considered when developing new communication protocols. If design principles are described informally it will be more difficult to reason about testability requirements as well as other aspects of protocol development. Some of the disadvantages of having informal design principles are: difficulty in deciding when to apply them; whether they can be applied successfully to a particular formal description technique or problem domain; whether they can be applied consistently to different formal description techniques and problem domains; and whether they can be used in combination with one another. Therefore, design principles should be formalized whenever possible although that is not enough by itself. It is necessary to establish a common representation to them so design principles can be formalized uniformly and related among themselves and to different FDTs in a consistent way.

Protocol design should be well structured. The list below describes some sound design principles for communication protocols. While they are not classified into categories they are related to distinct aspects of protocol design such as behavior, structure, and consistency.

It is important to note that testability requirements will be introduced into the design through these design principles. Certainly different testability issues will depend on different principles.

Behavioral and structural properties. The properties of a system can be broadly partitioned into behavioral and structural properties. Behavioral properties are related to operational characteristics, i.e., models to formalize all necessary aspects of communication system behavior. Important features of the model include how concurrent systems are specified (e.g., interleaving, true concurrency) and coordination (see Section 3.1.2). In the next section, this aspect will be further discussed.

Structural properties refer to the description of entities that comprise the system and their relationships with one another. Some of the structural concepts that should be formalized in any behavioral model are entities, interactions, interaction points, and system architecture.

Layering. This is one of the most widely-applied design principles in protocol design. The basic idea behind layering is to partition the overall system functionality into a hierarchy of layers. In this hierarchy, layer n provides services to layer $n + 1$ based on the cooperation of entities in layer n and the services provided by layer $n - 1$. Thus a layer hides the implementation details of the layers beneath it. This concept is central in the basic reference model for Open Systems Interconnection (OSI) [195].

Note that from the point of view of a user, the service is the important aspect whereas for the implementer it is the protocol.

Refinement. One of the most well-known approaches to system development is refinement. In this process a high level specification is transformed ideally by a sequence of correctness preserving refinements into an implementation.

Simplicity. The protocol should be based on a small number of functions that when put together are easy to understand. This is the case for light-weight protocols that are simple, robust, and efficient.

Modularity. The protocol should be built from a set of well-defined software functions. This leads to protocols that are easier to design, verify, implement, and test.

Multiphase protocols. Typically a protocol behavior can be partitioned into distinct phases, each one responsible for a specific task. In this way, each phase can be reasoned about individually according to the modularity principle discussed above.

Well-formed protocols. A protocol should be neither over-specified (in case of unreachable or non-executable parts) nor under-specified or incomplete (e.g., in case of unexpected events); it should be bounded (with respect to some limits), self-stabilizing (i.e., it is possible to reach a specific state and resume normal operation when started from an arbitrary state or in an abnormal situation), and self-adapting (it can adapt to some parameters).

Robustness. It should work under various workload conditions and different hardware/software platforms; it should guarantee compliance of real-time performance requirements depending on the application; it should be easy to modify when, for instance, a new technology is to be used.

Consistency. The protocol should exhibit some logical consistency properties such as safety and liveness. Besides that, performance properties should also be considered if real-time constraints are present.

Nondeterminism. Nondeterminism is a powerful tool in protocol design as explained in Section 3.1.1.

Some of these design criteria are examined in more details in [80].

3.2.3 Design process

The protocol development process follows the traditional software development cycle and can be divided in at least the following phases:

- *analysis*—functional and non-functional requirements of the protocol are identified;

- *specification*—a behavior description that incorporates the requirements is written;
- *design*—starting from the specification a high-level design is produced and refined until a low-level design is obtained.
- *implementation*—the low level-design is expressed in a programming language.
- *testing*—the code is tested for correctness with respect to the specification. Errors found in this phase may lead to revisions in the previous phases.

In the traditional protocol development process, implementation, testing, and debugging are often the focus of the development process. This is a situation found in other problem domains as well.

It is not difficult to understand the reasons why this is the case. Firstly, formal design is seldom used so that the implementation process is the first phase where a formal and unambiguous notation is employed, i.e., a programming language is used to represent the low-level protocol design. Secondly, the availability of tools such as compilers and debuggers in the implementation and testing phases but there is a general lack of sophisticated tools for design.

The consequence of this situation is that the later an error is found, the more expensive it is to fix it.

On the other hand, a formal design process means the introduction of principles, methods, and methodologies that will help a designer to build correct and reliable protocol specifications using a formal description technique. FDTs are essential in every phase of the protocol engineering process although, much more work needs to be done so they can be used to support the entire protocol engineering process. More specifically, an FDT should support the entire design trajectory including specification, verification, implementation and testing. The success in applying them to real applications will depend on the effectiveness that design steps are understood, how well FDTs support these design steps, and finally how convenient and effective are the FDT tools that implement them.

3.2.4 Requirements of an FDT

In general, FDTs provide a basis for: (i) development of unambiguous, clear and concise specifications, (ii) verification of specifications, (iii) functional analysis of specifications, (iv) development of implementations from a specification, and (v) determination that an implementation conforms to its specification (i.e., conformance testing).

To support (i), (iv) and (v), an FDT should satisfy the two basic requirements of expressive power and abstraction. Expressive power refers to the ability of an FDT to allow its users to compose, survey, understand, modify and extend a specification easily. This includes conciseness as well as proper structuring facilities for specifications, e.g., process composition, abstraction and instantiation, and recursion. The abstraction requirement refers to the ability of an FDT to specify the concepts of the application area, i.e., the OSI architectural concepts and constructions, at the appropriate level of abstraction. In this respect, OSI concepts (e.g., service access points, connection endpoints, service primitives, protocol data units and constraints) should be expressible in a completely implementation independent way without introducing a morass of arbitrariness and irrelevant details in the specification. To support (ii) and (iii), an FDT should have a strong mathematical basis that makes it easier to analyze and prove desirable syntactic and semantic properties of protocols.

These three requirements, i.e., expressive power, abstraction and strong mathematical basis, impose constraints on the specification, verification and testing phases. These constraints can be explained in terms of the gaps between specification and verification, and verification and testing.

The specification–verification gap. A specification formalism should be powerful enough to express different features, i.e., properties, constraints, and assumptions of different communication protocols, but at the same time able to specify a communication protocol succinctly. However, the more expressive a specification formalism is, the more difficult it is to define a simple and tractable mathematical model that can be used for verification purposes.

The verification–testing gap. In developing large systems, we cannot expect to be able to derive an implementation immediately from the initial specification S_0. Rather we expect the implementation phase to consist of a series of small and successive refinements of the initial specification until eventually an implementation I can be obtained. This process is called *Stepwise Refinement*, where a high level specification is transformed by a sequence of correctness preserving refinements into an implementation. The correctness is preserved in the sense that if S_{i+1} is the refinement of S_i, then any implementation correct with respect to S_{i+1} must also be correct with respect to S_i. This must be accomplished using some verification method. Thus a specification can be identified by a set of correct implementations. Consequently the implementation phase can be represented as a decreasing chain of sets $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_n$ with the final implementation I being a member of S_n. Furthermore, it is desirable to have the correctness of a specification S_i $(1 \leq i \leq n)$ determined in a compositional way, i.e., the correctness of the overall specification is dependent only on the correctness of its parts [1]. This leads naturally to decomposition of specifications into smaller and more manageable parts.

In practice, this refinement process is done with little or no verification at each step because of the specification–verification gap. Therefore, when an implementation is obtained, it has to be tested and its correctness cannot be fully guaranteed.

3.2.5 Comparing FDTs

Regarding the development and comparison of formal methods, Manna and Pnuelli [115] point out that "the activity of developing formal methodologies should consist not only of the construction of new models and the invention of new techniques for analyzing existing models, but also of a continuous assessment and comparison of models, both with one another and with the actual reality." For an interesting discussion about formal methods see, for instance, [41].

From the point of view of design for testability, the comparison and continuous assessment of formal description techniques are very important since testability requirements will be introduced during the design process and expressed using FDTs. In the following we cite different proposals to compare FDTs among themselves and to a specific problem.

Different proposals have been made that classify and compare formal methods. For example, Liu [112] gives a classification according to the models used which are state transition, programming languages, and hybrid. Avižienis and Wu [5] classify formal methods based on the approaches used: operational, definitional, and hybrid. Rozoy [147] gives a classification according to the models used namely, state-based, events-based, and words-based.

Another way of comparing FDTs is to evaluate their applicability to the same protocol specification so their strengths and weaknesses can be identified. In [16], von Bochmann specifies a simplified class 2 transport protocol in Estelle and Lotos and also presents a sketch in SDL. In [183], Vuong et al. specify an FDDI MAC layer protocol in Estelle, Lotos, and Communicating Rule System (CRS).

However, probably the most useful way of comparing FDTs is to compare their applicability to designing real communication protocols. In this way it will be possible to identify their limitations and propose solutions to overcome them.

In the context of ISO and ITU-T, three FDTs have been standardized. ISO (International Organization for Standardization) developed Estelle [86] and LOTOS [85], and ITU (formerly known as CCITT) developed SDL [27]. We note the standardized FDTs have been primarily developed and used in the specification phase of the OSI standards. In the following the major features of the three FDTs as relate to analysis, implementation and testing are briefly described.

Estelle. Estelle is basically an extended finite state machine (EFSM) model based on nondeterministic automata extended with the Pascal language and additional structuring constructs. It models a specified system as a hierarchical structure of automata module instances which may run in parallel, and may communicate by exchanging messages and/or sharing (in a restricted way) some variables.

Estelle separates the description of the communication interfaces from the description of the modules. All manipulated objects are strongly typed, i.e., detection of specification inconsistencies can be done at compile-time (static checking).

Estelle supports nondeterminism in a restricted way, namely at the level of activities (lowest level in the architecture hierarchy). In an execution step, a nondeterministic choice

is made among all "ready-to-fire" activities.

In Estelle, concurrency is implicit and can be synchronous or asynchronous (using the delay clause). Asynchronous parallelism between modules is only possible between subsystems, while synchronous parallelism is only possible within the subsystem, i.e., a module instance cannot run in parallel with its descendents. It is not possible to express interleaving directly. Estelle supports only asynchronous communication between modules. An interaction received by a module instance at one of its interaction points is appended to an unbounded FIFO queue associated with that interaction point. The FIFO queue can belong exclusively to that interaction point (individual queue) or can be shared with some other interaction points of a module (common queue).

LOTOS. LOTOS has a complete formal definition based on two paradigms: a transition system for expressing the dynamic behavior in the process part, and an abstract data type (ADT) definition based on ACT ONE in the data part. LOTOS models a distributed system as a process, which may be composed of many sub-processes. A sub-process is also a process. Thus, in LOTOS a system is specified as a hierarchy of processes. Actions internal to a process are unobservable. A process can interact with other processes which form its environment via observable actions.

In LOTOS concurrency is explicit and interleaving is easily expressed. LOTOS supports only synchronous communication between processes. All communication (interaction) with the environment is done through interaction points called gates. An interaction may occur only when two or more processes are ready to perform the same observable action. An observable action consists of offering/accepting zero or more values at a gate. An interaction may involve data exchange, and it is an instantaneous event, i.e., synchronous communication. It is possible to express process composition algebraically using the following operators: '|||' – interleaving; '|[]|' – synchronization; '| >' – disabling; and '>>' – enabling. Nondeterminism is a basic feature of LOTOS specifications and can be explicitly introduced by either the operator [] or by the internal event **i**.

SDL. SDL is similar to Estelle in that it is based on the EFSM model and allows different levels of abstraction. However, unlike Estelle it supports ADTs like LOTOS and has both textual (SDL–PR) and graphical (SDL–GR) definitions which can be mixed at the discretion of the user. SDL was developed by ITU and its formal syntax and semantics are described in ITU Recommendation Z.100.

SDL supports only asynchronous communication between processes. Every process has an associated input queue, which is similar to an individual queue in Estelle. The signals (messages) that the process is prepared to accept (i.e., to append to its input queue) are defined by a complete valid input signal set. In SDL, concurrency is implicit and asynchronous.

3.2.6 Application of FDTs

Some remarks on the application of formal description techniques are given below:

Expressing system properties. It may not be possible to express some system properties at the architectural level. These include requirements for performance, absolute time, cost, and other generic requirements such as type of traffic (e.g., stream oriented), fault-tolerance, or robustness.

Realizing design decisions. Design decisions taken during the design process may lead to solutions that are difficult or cannot be realized according to the syntax and semantics of the FDT. Therefore these decisions should be constantly evaluated when performing the design process.

In any case, the design process should always be performed keeping in mind the design criteria described earlier.

Specification style and levels of abstraction. The design process comprises different levels of abstraction. Depending on the level, an appropriate *specification style* should be used to reflect the desirable characteristics at that level according to some criteria. Of course, these styles need to be identified and compared for each FDT.

In the case of LOTOS, four specification styles have already been identified [172].

Low-level design and implementation. The design obtained at the end of the design process, i.e., a low-level design, is used as the basis for implementation which ideally should be done automatically or semi-automatically. Depending on the FDT used it may be difficult or even infeasible to implement certain aspects defined in the design as discussed in the next section. This problem is similar to the one described under the heading of "realizing design decisions" above, except that here the problem is between low-level design and implementation and not between different design levels.

The best solution in achieving a good implementation is to define precise mapping functions between constructs in the FDT and a given implementation language. The mapping should match the constructs in a natural way.

Semantic models. Differences in the semantic models of FDTs have direct consequences in the way architectural and behavioral concepts in communication protocols are formally described. This fact triggers a chain of consequences: different specification styles that lead to differences in the appearance of formal descriptions which in turn pose interpretation questions to implementers and testers. This is discussed in more detail under the heading of "semantics aspects" in the next two sections.

Adequacy of FDTs. It seems that there is no single FDT that is completely adequate for all problem domains despite the fact that FDTs have shown to be very useful in the development of a broad range of systems. In fact this observation seems to be the main reason that motivated the ISO workgroup on FDTs to create two subgroups to develop two FDTs based on different paradigms in early 1980's. These subgroups eventually developed Estelle and Lotos [173].

3.3 Implementation

A protocol implementation should satisfy the requirements presented in the protocol specification. In addition, some constraints may be imposed on the implementation by the language used to implement the protocol, the environment in which the protocol will run, implementation decisions, etc. Some of the implementation decisions may have to do with

the number of simultaneous connections, expected performance of the implementation, and other issues specific to the protocol.

Based on the requirements and constraints, a protocol implementation is often derived through a refinement[3] process. Therefore, the implementation phase is directly affected by the design phase.

Ideally, both the specification and the constraints should be written using the same FDT so they could be possibly translated through an automatic or semi-automatic process into a programming language.

3.3.1 Meaning of implementation and testability

In the following, specification and implementation will be related by the chain of sets $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_n$, where S_0 is the initial specification. Therefore, specification and implementation are relative notions in this chain. There are several relationships between implementation and specification which are discussed below [21]:

(i) Implementation as a *reduction* of a specification. Set S_i $(1 \leq i \leq n)$ is said to be an implementation of S_{i-1} if S_i is obtained from S_{i-1} by resolving *choices* existent in S_{i-1}. In other words, a behavior is valid for S_i only if it is valid for S_{i-1}. An invalid behavior for S_i should also be invalid for S_{i-1}. This is the case of reduction of nondeterminism in CSP (Communicating Sequential Processes) [22].

(ii) Implementation as an *extension* of a specification. Set S_i $(1 \leq i \leq n)$ *adds information that is consistent* with S_{i-1}. In other words, a valid behavior for S_{i-1} should also be valid for S_i, which can do more but should be consistent with S_{i-1}. An invalid behavior for S_i should also be invalid for S_{i-1}.

(iii) Implementation as a *refinement* of a specification. Set S_i $(1 \leq i \leq n)$ *provides more detail* about S_{i-1}. In other words, both S_{i-1} and S_i are *extensionally* equivalent in the sense that their observed behaviors cannot be distinguished. However, the *intention* of S_{i-1} and

[3]In this thesis, we use the term refinement to mean a decomposition process that includes the principles described in Section 3.3.1.

S_i is not the same since S_i gives more details about the intended structure of S_{i-1}. This can be seen as a transformation. In process algebra, this notion of implementation was introduced by CCS (Calculus of Communicating Systems) [128].

It is clear that the decomposition process of a protocol specification can affect the testing phase. Depending on how the decomposition is done, the implementation may be easier or harder to test. The important point is that the implementation design given by S_n should satisfy both the protocol specification and testability requirements that were introduced in S_i $(1 \leq i \leq n)$.

In this chain, S_n can be obtained through a combination of reduction, extension and refinement. The principle to be used at each step of this process will depend on each specification issue.

3.3.2 Implementation issues that affect testability

This section discusses issues that affect testability when deriving an implementation from a formal specification. First, the issue is discussed and then possible improvements are presented. Some of the issues discussed below could be grouped under a single heading. However, we prefer to put them apart so they could be emphasized.

- **Refinement of behavioral and structural properties**

Issue. A protocol specification defines behavioral and architectural properties such as interactions (events/actions), data, and interaction points (interfaces) that need to be reflected in an implementation. However, it is not necessary to have a one-to-one relationship between a property in the specification and in the implementation. For instance, in general the granularity of specification events/actions are coarser in the specification than those in the implementation. Some typical specification actions are sending or receiving a message, and starting a timer. On the other hand the execution of each computer instruction can be seen as a distinct implementation action. Similarly, events in the specification can be modeled in the implementation as several actions or several distinct "smaller" events.

Another example is that the granularity of data in the specification is often coarser than

that in the implementation. Typically, a piece of data in the specification is a message, while at the implementation level, it may be a computer word.

Testability. It is very important to define mapping functions for behavioral and architectural properties. These functions should be based on a semantic model to be used by different FDTs. Furthermore, these mapping functions should consider the expressivity of the specification and implementation languages as discussed below.

Independent of the semantic model used, it seems that the granularity of events/actions and data in the specification should be as close to the granularity of events/actions and data in the implementation as possible. Otherwise the testing process will have to consider the components or parts that implement events/actions and data.

- **Expressivity of specification language and implementation language**

Issue. The application scope for all FDTs should not be restricted to the specification phase of a communication protocol. The goal of all FDTs is to support the entire protocol engineering cycle. As an example, the Lotosphere project [98] has shown the potential of applying LOTOS to the entire system design and implementation trajectory. Nevertheless, there are some issues that need to be addressed:

▷ Expressivity affecting the decomposition from specification to implementation:

When mapping a formal specification to an implementation, depending on the expressive power and abstraction level of the formal method it may not be possible to implement some aspects of the specification. For example, using the infinite summation construct in LOTOS, behavior expressions can express solutions to undecidable problems [54]. As another example, in both Estelle and SDL, message channels between processes are unbounded by definition. Some safety properties have been formally proven to undecidable for unbounded systems [144].

▷ Expressivity affecting the representation of concepts in the specification:

Depending on the target platform and implementation paradigm, it may be difficult to implement or represent certain aspects of the specification such as nondeterminism, communication, synchronization and timers. In the case of channels in Estelle and SDL, no

implementation could truly implement unbounded message channels since computers have finite memory resources.

Testability. The applicability of each FDT will depend on its expressive power, abstraction level and other criteria discussed in Section 3.2.3. The FDTs were designed to be implementation independent [173]. However, in order to derive concrete implementations from abstract specifications, it is important that designers know what features of the languages will pose problems to implementers. Once designers have more experience applying formal description techniques to specifying communication protocols and distributed systems, it will be easier to identify in each FDT the desirable characteristics that will ease the testing process.

Furthermore, once these features are identified it is very important to have tools that will help designers to introduce testability in the design according to the principles established in Section 1.4. In [32] a survey on tools used for protocol development based on FDTs is presented, and none of them have facilities to incorporate testability into the design.

Another option to the decomposition process is to use a *two-tiered* approach to specification such as in the Larch project [73]. Each Larch specification has components written in two languages: one designed for a specific programming language called Larch interface languages, and another common to all programming languages called Larch shared language. A key aspect in this approach is a common semantic model to be used by the Larch interface languages.

- **Semantic aspects**

Issue. Because of the differences in semantics for different formal methods, the same issue can be handled in different ways by different formal methods. In the following, some examples are presented.

For example, LOTOS handles events that are invalid by forcing a deadlock situation, whereas in Estelle and SDL events that are not valid are simply discarded. Therefore it may not be simple to describe a "robust" implementation. In Section 3.4 this particular problem of handling events is further discussed.

Another example has to do with how interactions take place between entities and the

environment. In LOTOS the occurrence of an interaction affects all processes involved immediately including the environment. In Estelle and SDL, there is an input/output asymmetry caused by the queues, i.e., interactions are received from queues and sent to queues. This fact is reflected in the different views that peer entities and the environment take. More specifically, the output produced by a process affects the process immediately, whereas the environment and the peer entities are affected later when both are ready to accept that interaction.

Testability. All semantic aspects in the formal method that may not lead to "robust" specifications should be identified and dealt with.

- **Relation between specification style and implementation**

Issue. There are two points to be considered:

▷ Current specification styles:

Vissers, Scollo and van Sinderen [172] have identified some specification styles for LOTOS, i.e., monolithic, state-oriented, constraint-oriented, and resource-oriented, which can be applied for other formalisms as well. These styles are mainly related to the structure of the specification itself. As pointed out by van Eijk, Kremer and van Sinderen [55], each style imposes a different degree of difficulty for the implementation and testing processes.

▷ Testability specification styles:

After a testability requirement is identified, it is necessary to establish a development process in order to use it in the design. Certainly this process will depend on the requirements and will lead to different specification styles. It seems there will not be just one but several testability specification styles.

Testability. The first point above shows that a specification should be written in a style that is easy to implement and test. In this case, style transformations should be formally defined and, ideally, should be automated.

The second point shows that a testability specification style is dependent on the testability requirements. Since communication protocols are being developed for new applications

and environments it seems that there will be different testability specification styles.

- **Lower-level design and implementation decisions**

Issue. After obtaining the lower-level design, a number of implementation decisions have to be made that will affect the testing process. In the following, these issues are discussed:

▷ *Realizing choices present in the specification:*

A formal specification should not describe implementation details. For instance, at the specification level it is possible to have a situation where two or more events can occur but the semantics of the formal method should not say which event will be treated first. At the implementation level this decision will have to be made. It may be the event that occurred first, or a specific event according to some criteria, or simply an event chosen at random. In the context of the OSI conformance testing methodology and framework [87], that information is given in the PICS (protocol implementation conformance statement) and PIXIT (protocol implementation extra information for testing).

Also related to this issue is the question of physical limits such as lengths of queues that are often not restricted in FDTs. Of course, physical limits will have to be imposed to the implementation.

▷ *Internal structure:*

At a lower-level some design decisions should be made when defining an internal structure for the implementation. For instance, single- or multi-layer protocols can be decomposed into different functions or phases. These distinct logical parts can be implemented through separate modules or concurrent operating system tasks, in a single or multiple execution environment, or less appropriately a monolithic code, or even in a front-end processor.

Furthermore, in layered communication architectures, it is natural to map a protocol layer to a process in the implementation. This implementation strategy is frequently criticized as imposing substantial overhead and therefore highly inefficient [138, 166]. From the point of view of DFT, the important aspect in the decomposition is to take the testability

requirement to the final implementation.

▷ *Parts comprising the implementation:*

The lower-level design is usually comprised of two parts: a machine-independent part that realizes the communication protocol, and a machine-dependent part that includes aspects of memory management, interprocess communication, and mechanisms for event handling. This latter part is dependent on the machine architecture and the host operating system.

In the case of black-box testing, test cases are generated considering only functional requirements, i.e., the machine-independent part.

Testability. An implementation design should incorporate both the protocol specification requirements and testability requirements. Preferably, different protocol functions should be implemented by different modules following the basic principles established in software engineering. Otherwise it may be difficult to test all protocol functionalities.

This means that choices made at the implementation level should be "visible" during the testing process. A possible way of accomplishing this is the use of a testing interface as discussed in Section 2.5.6.

- **Quality of code**

Issue. The problem of quality of the implementation code is not particular to communication protocols but common to all application domains.

Testability. Ideally, the implementation should be written according to principles defined in software engineering such as modularity, separation of concerns, anticipation of change, generality, and incrementality. These principles are discussed in detail in books on software engineering. For a reference see [66].

- **Development environment**

Issue. As explained above under the factor "Lower-level design," a protocol can be implemented in different ways: monolithic code, separate modules, or concurrent operating system tasks, in a single or multiple execution environment. In the case of monolithic code

a single environment is used. In the other cases, different development environments can be employed. When putting the modules together, it may be difficult to perform integration testing and debugging.

Testability. The development process can be facilitated if there is a uniform distributed implementation environment. That environment should support a single formal description technique and implement the specification presented in that FDT automatically or semi-automatically.

Good software tools have demonstrated to be very important for development of more reliable communication protocols. In [32], Chanson, Loureiro, and Vuong discuss several important characteristics that an FDT tool should have.

In general, tools that support Estelle and SDL are not difficult to obtain since processes and modules communicate through message passing which are naturally mapped to tasks in one or more computers that also communicate through message passing. For LOTOS, the situation is more complicated since two or more processes can participate in a rendezvous interaction.

3.4 Testing

In this section we briefly describe several concepts that are important in the area of protocol testing and then present the issues that affect the testing process. Some possible improvements are also discussed.

3.4.1 Overview of protocol testing

Service primitives and service access points. In protocol testing, test cases are applied to the protocol implementation through service access points. These test cases represent service primitives defined in the protocol specification. Therefore service primitives and service access points play a key role in protocol testing.

Types of protocol testing. In software testing we can distinguish between functional testing (also called black-box testing) and structural testing (also called white-box testing). In functional testing the system is treated as a black box and its functionality is determined

by its external behavior (no assumptions are made about its internal structure). Structural testing is based on the internal structure of the system and the goal is to test the program code thoroughly.

In the following, we present informal definitions for some types of protocol testing which have different goals.

Diagnostic testing. This is typically inhouse testing where the tester is usually the implementor with complete access to the internals of the implementation, i.e., white-box testing. The goal is to ensure the IUT is bug-free and works correctly, but not guarantee its conformance to its specification.

Conformance testing. The protocol implementation is tested with respect to its specification. The goal of conformance testing is to check whether the external behavior of the protocol implementation follows the behavior defined in the protocol specification. Conformance testing is a type of functional testing.

Interoperability testing. Conformance testing does not guarantee successful communication between systems. The reason is that a protocol specification often provides different communication functions with different mandatory and negotiable options, and alternative features in terms of behavior, parameters, and quality of service (e.g., addressing information, timeout intervals, and maximum number of connections supported by a connection-oriented protocol). Therefore, protocol implementations need to be tested in a real environment. The goal of interoperability testing is to check whether a protocol implementation interacts properly with other entities in a real environment.

Performance testing. The goal of performance testing is to measure quantitatively the performance of a protocol implementation. This type of testing is very important for real-time applications.

Robustness testing. The goal of robustness testing is to analyze the behavior of a protocol implementation in an erroneous behaving environment. This type of testing is very

important for fault-tolerant systems.

The most studied type of protocol testing has been conformance testing since this is the very first type of testing that needs to be done after diagnostic testing. Furthermore, conformance testing of different implementations derived from the same protocol specification should provide identical conformance test results. In order to achieve that, protocol testing should be based on well-founded principles, and use accepted tests and results. ISO together with IUT have developed a standard for conformance testing of Open Systems known as the ISO IS9646 or the OSI conformance testing methodology and framework (OSI-CTMF) [87].

Testing activities. The testing process involves some activities that are common to all types of protocol testing discussed above. The activities include:

Test generation. The process of deriving test cases from a protocol specification.

Test verification. The process of checking whether the behavior expected from a test case when applying to an IUT is in fact valid with respect to the specification.

Test selection and parameterization. A protocol implementation does not need to support all functions and features present in the specification. Therefore, only a subset of the possible test cases may be applied to the implementation. The process of identifying the correct subset of test cases is called test selection. Test parameterization is the process of assigning values to the parameters of the test cases.

Test execution. This is the process of generating an executable test case from a selected and parameterized test case and applying it to the implementation. There should be a way to record the history of the test execution for posterior analysis.

Test methods. ISO IS9646 (OSI-CTMF) [87] has defined four abstract test methods: local (LTMs), distributed (DTMs), coordinated (CTMs), and remote (RTMs) (Figure 3.1). All of them use a point of control and observation (PCO) below the implementation under

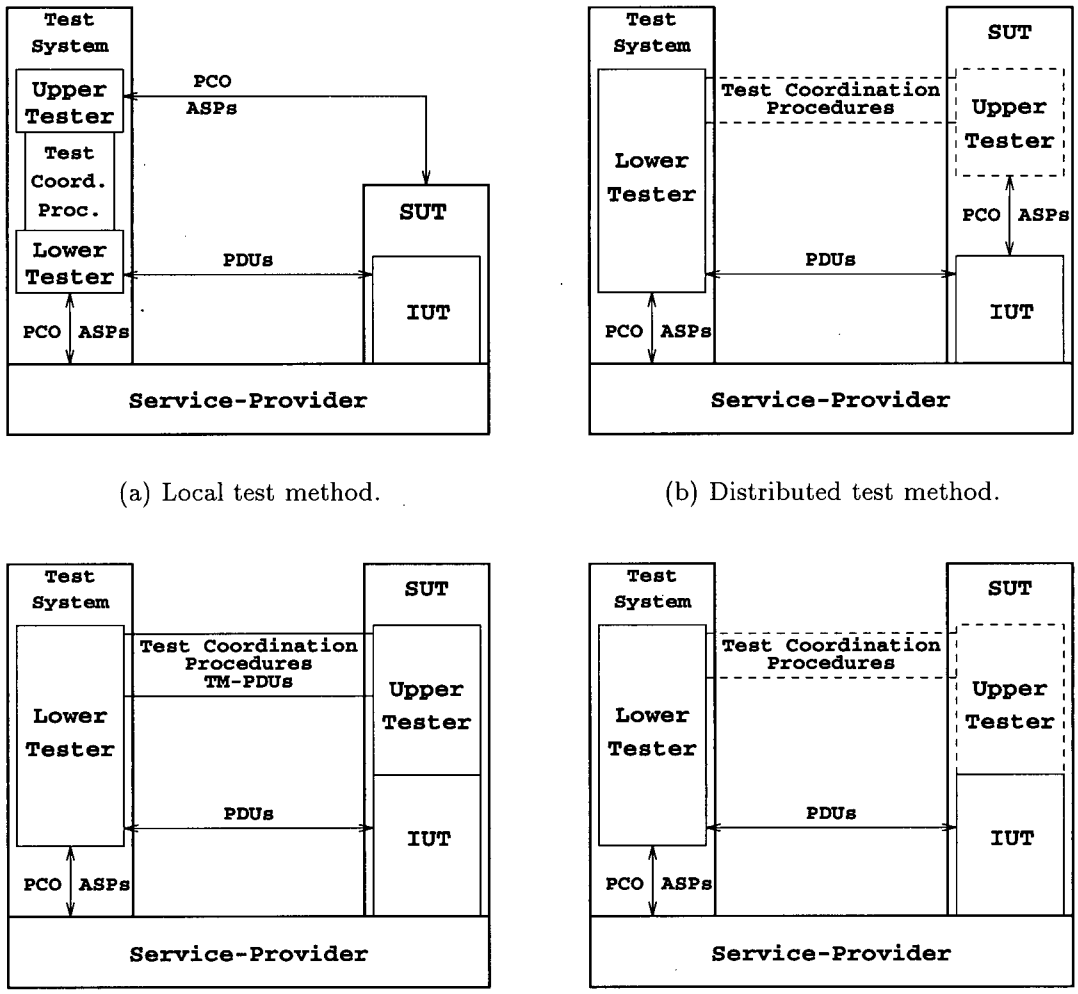
test (IUT) where abstract service primitives (ASPs) are input and output. Also, a lower tester separated from the system under test (SUT) exchanges protocol data units (PDUs) with the IUT. In the following, we give a brief overview of each abstract test method:

Local and distributed test methods. Both use a point of control and observation for the ASPs (Abstract Service Primitives) at the upper service boundary of the IUT in addition to the lower boundary. In the LTMs the upper tester is located within the test system whereas in the DTMs is located within the SUT. The LTMs require the upper service boundary of the IUT to be a standardized hardware interface. The DTMs require it to be either a human user interface or a standardized programming language interface. In both methods, it is required to access this interface for testing purposes. In the LTMs, the test coordination procedures are realized entirely within the test system. In both methods, the requirements for the test coordination procedures are specified, but not the procedures themselves.

Coordinated and Remote test methods. Both test methods do not require access to the upper service boundary of the IUT. In the CTMs, the test coordination procedures are realized by test management protocols (TMPs). The upper tester is an implementation of the relevant TMP. In the RTMs, the abstract test suite (ATS) may imply or informally express requirements for test coordination procedures although, it is not possible to make any assumption regarding their feasibility or realization. There is no upper tester but some of its functions may be done by the SUT.

Figure 3.1 depicts test methods for single-layer IUTs. Variants of these methods can be applied to multi-layer IUTs, where an IUT is tested layer by layer. An embedded method is a special case of multi-layer testing where a lower layer protocol is wrapped around upper layer protocol data units (PDUs). In this case, we are only interested in testing the lower protocol that is accessible through the upper layers.

The Ferry Clip method is an approach that realizes any of these abstract test methods [33, 194].



(a) Local test method.

(b) Distributed test method.

(c) Coordinated test method.

(d) Remote test method.

Figure 3.1: Test methods.

3.4.2 Testing issues that affect testability

This section discusses issues that affect testability for each concept in protocol testing presented above. It follows the same format used in the previous section, i.e., first the issue is discussed and then possible improvements are presented.

3.4.2.1 Services and service access points

Vissers and Logrippo [174] discuss the importance of the service concept and related topics (e.g., service primitives, primitive parameters) in the design of communication protocols. In [69] Gotzhein discusses the concept of interaction point (IP) and suggests a list of properties

an IP should have. In [140] Probert and Saleh present a survey of synthesis methods of communication protocols including methods that use the service specification as the starting point in the synthesis of the protocol entity specification (or procedure rules).

- **Interface refinement**

Issue. A protocol specification should specify the behavior of a protocol entity at both the upper and lower service access points, i.e., (N)-SAPs and (N-1)-SAPs. At the implementation level, two or more SAPs may correspond to one SAP at the specification level. Therefore, when testing a protocol, it is necessary to relate the SAPs in the implementation to those in the specification.

Testability. If a SAP at the specification level is represented by two or more SAPs at the implementation level we may have a nondeterministic behavior of the IUT due to non-observability. This is due to the fact that when events or actions are offered at two or more SAPs it may not be possible to order them and predict whether the behavior is correct or not.

It should be easier for the tester to check whether the implementation's behavior follows that of the specification if a one-to-one correspondence is kept between SAPs at the specification and implementation levels.

- **Expected behavior from the environment**

Issue. It is difficult if not impossible to have an implementation that can work properly in an environment with completely arbitrary behavior. Furthermore, a protocol specification cannot anticipate all possible environments in which a protocol will run and enumerate all possible behaviors. In practice, protocols are specified according to the applications they will support (e.g., real-time, fault-tolerance).

Testability. The key point is to determine appropriate constraints based on the application the protocol will support. Note that some commonly accepted behavior of the environment is to be expected. For example, an environment should not change or even access the addressing space of the running system (assuming a multi-tasking operating sys-

tem). On the other hand, the environment should not be inhibited from raising an exception related to the implementation when it occurs.

- **Handling unexpected events**

Issue. A formal protocol specification should anticipate the events offered by the environment at the SAPs and their corresponding responses. As seen above this is very difficult to accomplish. The events that are not considered by the specification should be handled at the implementation level according to the semantic model of the formal method from which the implementation is derived. This may lead to different behaviors since the semantic models of the formal methods are not identical. For example, in LOTOS an invalid event is required to lead to a deadlock whereas in Estelle and SDL, invalid events are discarded.

Testability. For testing purposes, the OSI conformance testing method and framework distinguishes between valid, inopportune, and invalid events.[4] Implementations of the same communication protocol derived from formal protocol specifications written in different FDTs should exhibit consistent behaviors when offered the same event in the same state. This is necessary for different implementations to interoperate. Therefore some kind of "event consistency" is required. The idea is to check and transform the event before it is offered to the module that will handle it. If the event is invalid it could be replaced by an appropriate one expected by the module and therefore invalid events would be handled properly.

- **Relating events**

Issue. When testing a protocol implementation, we have to deal with the different notions of event defined by the specification, test suite, and implementation. A formal protocol specification defines the observable system behavior in an abstract manner by way of interactions (events) at service interfaces. Test suites described in TTCN [141] and ASN.1 [133] define the sequences of abstract events that should be observed during test

[4]Let e be an event that happens in state s. Event e is *valid* if state s specifies how the system should react to it. Event e is *inopportune* if state s does not specify how the system should react to it but it is *valid* in another state s'. Event e is *invalid* if there is no state s that specifies how the system should react to it.

execution. In a protocol implementation, ASPs can be realized by procedure calls, message exchanges, or any other appropriate mechanism provided by the implementation language. The invocation of these mechanisms leads to interactions (events) at the implementation level.

The problem is that there is no clear relationship between specification events, abstract test events, and implementation events. Therefore the mapping process is not clear and depends very much on human understanding.

Testability. It is necessary to define a formal mapping between events in the specification, test suites, and implementations. This is an important problem since conformance testing is based on externally observable events. Meanwhile the mapping description should be documented as part of the PICS and PIXIT information.

3.4.2.2 Types of protocol testing

In Part 1 of the document that describes the OSI conformance testing methodology and framework [87] different types of protocol testing are discussed. However, most of the testing methods proposed so far are related to conformance testing and much more work needs to be done on the other types of protocol testing.

- **Testing requirements**

Issue. Ideally, testing requirements should be defined only in terms of functional behavior (black-box). However, it may be very difficult to test some conformance requirements without considering white-box testing (see, for instance, the issue "lower-level design" in the previous section). Therefore, not all test cases may cover all conformance requirements in a protocol specification.

Testability. In order to be able to test the conformance requirements of a protocol specification some combination of black-box and white-box testing will be needed. Furthermore,

new methods for interoperability, performance, and robustness testing are needed.

- **Conformance testing and the environment**

Issue. The goal of conformance testing is to check whether the external behavior of the protocol implementation observed at external interaction points follows the behavior defined in the protocol specification. This definition of conformance relates two descriptions at different levels of abstraction, i.e., specification and implementation, independent of the environment.

In fact, formal definitions of conformance proposed in the literature have generally followed this framework. See for instance, observational equivalence between CCS processes [128], the *satisfy* relation between a CSP process and a formula of trace logic [77], and the *conf* relation and testing equivalences for LOTOS processes [20].

Suppose the definition above is modified to consider the behavior of the specification S and implementation I according to a given environment E. It would probably be very difficult to establish conformance between S and I since the set of all possible environments E can be very large. This problem is present in other types of testing as well.

Testability. Of course, if the environment E is known in advance, it is possible to restrict the definition above and, therefore, establish a conformance relation between S and I to be used during the testing process (for instance, for test case generation). Although, for open systems this is not the case in general since it is difficult to anticipate all possible environments in which I may run.

It seems that there are two ways to tackle this problem. First, to study whether it is possible to define conformance relations independent of the environments. Second, to identify and classify environmental properties that are important to conformance relations. In Chapter 4 we discuss further this problem.

3.4.2.3 Testing activities

Protocol testing activities have been studied extensively in the literature, including test generation [19, 44, 124, 158]; test verification, selection and parameterization [131]; and test execution [8, 153]. Overviews protocol testing can be found in [131, 157].

A. *Test generation*

- **Assumptions about the specification**

Issue. Most of the current methods on test cases generation assume the specification is given as a completely specified deterministic finite state machine. Furthermore, the methods in general cover only the control aspects of the protocol. Other important aspects such as nondeterminism and data aspects are often not covered.

Testability. In order to produce effective test cases, all aspects of a protocol specification should be considered during test case generation. Some work has been done along this line but further research is needed. Only recently has some work been done in test generation and fault diagnosis for protocols modeled as nondeterministic finite state machines [3, 63, 65, 95, 114, 94], and considering both the control and data parts [34, 124].

Regarding data aspects, the Abstract Syntax Notation One (ASN.1) is a notation for describing data structures similar to data type definitions available in programming languages such as Pascal, C, or Ada. ASN.1 can be combined with an FDT in at least the following three scenarios [133, 170]:

1. translation of ASN.1 definitions into the corresponding FDT data structure definitions and vice-versa;

2. enhancement of an FDT with ASN.1 definitions;

3. replacement of the FDT data structure definitions by ASN.1.

The first scenario is currently most popular. The second one is an extension of the FDT. And the third one could be used for defining test case generation methods for the data part of the protocol. In this case, the same algorithm could be applied to different FDTs since the data part would be expressed in the same way. This would improve testability in relation to test case generation.

- **Physical interface used during the testing process**

Issue. The physical interface used in the testing influences the process of test case

generation. Some details of the test case might change because of this interface. For instance, if some timing constraints need to be tested, the test case has to consider the delay introduced by the service provider.

Testability. Aspects that are interface dependent need to be identified during the generation of abstract test cases. Depending on the actual interface, these aspects should be substituted accordingly during the generation of concrete test cases. There is also the case that some test cases may no longer be valid. For example, a test case that uses the upper tester to check a test requirement can no longer be applied if the upper tester is not present in the test method.

- **Handling unexpected events**

Issue. The process of generating test cases from a formal specification should consider the problem of handling unexpected events. Otherwise only events mentioned in the specification could be handled.

Testability. A possible solution to this problem is to consider two types of events: valid (as defined before) and unspecified (comprising inopportune and invalid events as defined before). Therefore test cases which cover unspecified events could be generated depending on some criteria or constraints on the specification.

B. *Test verification*

- **Generation process**

Issue. Test verification is mainly required for manually derived test cases. Test verification can also be used to validate test cases generated from other specifications. This could be used to check verdicts for similar test cases derived from different specifications which may or may not be written in the same specification language.

Testability. To avoid possible errors in assigning verdicts to test results, test cases should be generated automatically from a formal protocol specification. If performance issues also need to be tested then a timed test case verification model should be used [132].

C. *Test selection and parameterization*

- **Representative set of test cases**

Issue. The set of valid, inopportune, and invalid interactions or events that can be applied to a protocol implementation is in general extremely large. Due to time and cost constraints, some strategy to select test cases is necessary.

Testability. It is necessary to establish criteria based on coverage to select a meaningful set of valid, inopportune, and invalid test cases. Vuong et al. [42, 119, 181] have presented a metric theory of test selection and coverage for communication protocols but further research is needed in this area.

D. *Test execution*

- **Transient states**

Issue. Suppose that the IUT is in state s_i at a given moment in time. If the IUT moves to state s_j without an externally controllable stimulus, state s_i is said to be a *transient state*.

Testability. Transient states make testing more difficult since the tester cannot predict when the IUT will leave a transient state and enter a controllable state. Therefore transient states should be avoided in protocol design.

- **Test synchronization**

Issue. The test synchronization problem addresses the coordination or synchronization between the lower and upper testers during test execution. This is an important issue for all test methods presented in Section 3.4.1, and the Ferry Clip approach [33, 194] is designed to alleviate this problem.

Testability. The synchronization problem has been studied in [8, 153] but further research is needed for multi-layer and embedded testing using different test architectures.

3.4.2.4 Test methods

The standardized test methods are defined in [87]. For an overview of the test methods see [108, 109, 152].

- **Applicability**

Issue. According to the test methods presented in Figure 3.1 we can see that the SAPs are not always directly accessible to the tester. The points where the tester controls and observes the IUT are called points of control and observation (PCOs). Therefore which test method to use will depend, to a large extent, on the characteristics of the SUT.

Related to this issue is the fault detection power associated with each test architecture which was studied in detail by Dssouli and von Bochmann [53].

Testability. The local test method should be applied when the SUT has two hardware interfaces. The distributed test method needs an upper interface that can be accessed by a user or a standardized programming language interface. The coordinate test method may be applied if the upper tester implements a standardized TMP. Single-layer test methods are the most popular for testing the majority of the protocol conformance requirements.

- **Interface refinement**

Issue. Abstract test cases are generated to access services at SAPs as defined in the specification. However, it may be the case that two or more SAPs at the implementation level may correspond to one SAP at the specification level as discussed in Section 3.4.2.1. Consequently there is no way to map test cases to the appropriate SAPs in the implementation without further knowledge on the purposes of each SAP.

Testability. Before generating concrete test cases from the abstract test cases, it is necessary to define a mapping process between SAPs in the specification and implementation so that test cases can be applied to the correct SAPs in the implementation. To automate the mapping process, it is necessary to define some mechanism to express the function of each SAP at the implementation level.

3.5 Conclusions

To ease the testing of a software or hardware system, an important requirement of the design should be testability. In fact, in some areas such as VLSI design, research activities have been carried out in this direction (i.e., design for testability) for a long time. In this case, proposed testing techniques can be classified as *a priori* approach, since the specification is designed to meet some testing requirements.

The study of testing of distributed systems in general, and communication protocols in particular using a priori approach is a relatively new research area. Most of the work that has been done in DFT in the software domain is related to the structural design and we believe that other aspects need to be considered as well.

In this chapter we have introduced a framework for the study of design for testability. A number of problems that affect the testing process were identified and possible solutions to improve testability discussed (Sections 3.1–3.4). This framework can also be used to reason about testability as of one the criteria for comparing formal methods, which was presented but not elaborately discussed by Larsen [103] (see Section 2.6.3.1). It is also interesting to note that at the design level it seems to be more difficult to separate the issues that affect verification and testing. For example, the nondeterministic choices that may be introduced in the specification can affect verification (in terms of the number of states to be searched) and testing (in terms of the number of correct behaviors to be tested). Therefore many issues discussed in this paper might be applicable to verification as well.

As mentioned in Section 2.8, the process of designing communication protocols should comprise synthesis and analysis techniques.

It is hoped that the framework introduced here will facilitate and encourage further research activities in formalizing and solving the DFT problems, thereby alleviating the testing process for communication protocols. To our knowledge this is the first study that provides this general treatment to the problem of designing communication protocols with testability in mind.

Finally, based on the framework two definitions are given below:

- *DFT-oriented design specification*—a specification that has incorporated some testability

requirements.

- *Testability-oriented design process*—a design process that preserves the testability requirements introduced at a given phase and which are carried to the testing phase.

Chapter 4

Self-stabilizing protocols and DFT

According to the conformance testing methodology and framework (CTMF) [87] a pass verdict should be assigned to a test case if a given event e is valid for the current IUT state s. By valid we mean that e satisfies all testing requirements as defined in the specification. Unfortunately, there is an important class of errors, namely coordination loss,[1] that cannot be anticipated because some source of error is *external* to both the tester and the implementation under test (IUT). Furthermore, it is not possible to simulate their occurrence exhaustively in the test environment. Therefore they are very difficult to catch in the testing phase. In this chapter we assume that a message with an invalid checksum can be detected by a lower level protocol.

In this chapter we propose using self-stabilization as a design principle to overcome this testing problem. We present a set of novel design principles to design self-stabilizing protocols, give an example of a self-stabilizing protocol, and define two new relations based on an environment that exhibits coordination loss. The chapter is organized as follows. Section 4.1 describes the problem in the context of conformance testing. Section 4.2 discusses the important points that should be followed when developing a new DFT technique. Section 4.3 presents a brief overview of self-stabilization. Section 4.4 describes the formal model used in this chapter. Section 4.5 presents a set of design principles to design self-stabilizing protocols. Section 4.6 gives an example of a protocol that is not self-stabilizing and its self-stabilizing version using the design principles described earlier. Section 4.7 presents a new conformance relation based on the external behavior. Section 4.8 discusses the related

[1]Some errors that may cause coordination loss are inconsistent initialization, process failure and recovery, and memory crash. These errors are explained in more detail in Section 4.3.

work. Finally, Section 4.9 presents the conclusions for this chapter.

4.1 Conformance testing and problem statement

The ultimate goal of OSI is to allow the interconnection of different systems that follow the same set of standards. In a real open system it is common to have the same set of protocols implemented on different architectures and environments by different people. These implementations must be tested for conformance to the specifications.

Conformance testing, as discussed in Section 3.4, is one of the basic activities in the protocol development process. The conformance testing and methodology framework (CTMF) identifies two types of conformance requirements: static and dynamic. Static requirements are related to functions requirements. Dynamic requirements refer to the dynamic behavior of the system. There is also a protocol implementation conformance statement (PICS) produced by the implementor describing the functions and options present in the implementation, and the protocol implementation extra information for testing (PIXIT) that provides specific information for testing purposes.

According to this framework, an implementation conforms to its specification if it satisfies both the static and dynamic conformance requirements, and is consonant with its PICS. This should be demonstrated through one of the four test methods defined in the CTMF (see Section 3.4.1).

Note that if a protocol specification S contains an error and an implementation I faithfully implements S then I conforms to S. Of course, if the error is identified in one of the protocol development phases, i.e., specification, implementation, testing, or even after that, the specification must be revised and all changes made in S should be reflected in I accordingly.

In practice, when I is derived from S it is assumed that the specification is correct. The fundamental point in this statement is the meaning of correctness. The specification may be only correct with respect to a specific property such as the absence of deadlock although there are other protocol properties that should be checked. As mentioned earlier, due to the complexity, there are limitations in this process and different solutions have been proposed. For instance, some alternatives to exhaustive reachability analysis — the

technique used in most automated validation systems — are based on search heuristics, hashing techniques, and reduction methods. For an overview of validation methods, their limitations and alternatives see for instance [79, 80].

In Figure 4.1 we show a partial view of the protocol development process. It is clear that

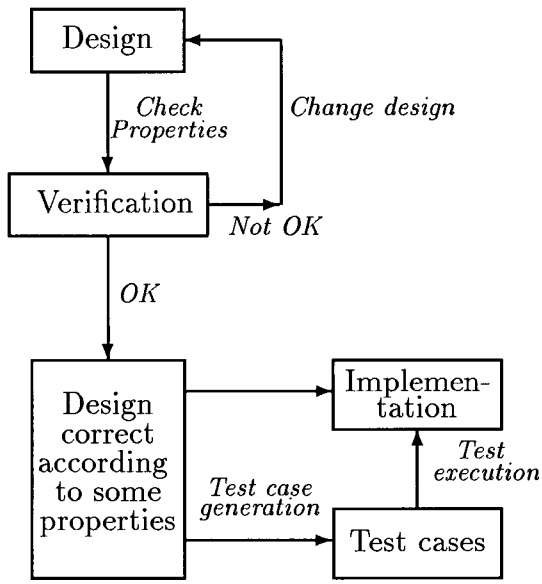


Figure 4.1: Partial view of the protocol development process.

in the protocol development process as is in software in general, verification and testing are complementary and mutually supportive techniques.

Recall from Section 3.4.1 that there are different types of protocol testing: diagnostic, conformance (C), interoperability (I), performance (P), and robustness (R). From their definitions it is clear that they have different testing goals (or objectives) and this is represented in Figure 4.2.

Basically, Figure 4.2 shows that when performing interoperability, performance, or robustness testing, the IUT should conform to the specification and also to specific testing goals defined by each type of testing. Diagnostic testing is not shown in the figure because it is often done in-house and is the first type of testing performed. In particular, robustness testing can also be considered when performing interoperability or performance testing.

As explained earlier, coordination loss defines an important class of errors [71, 80, 100, 150, 162] which are very difficult to catch by conformance testing. The conformance testing

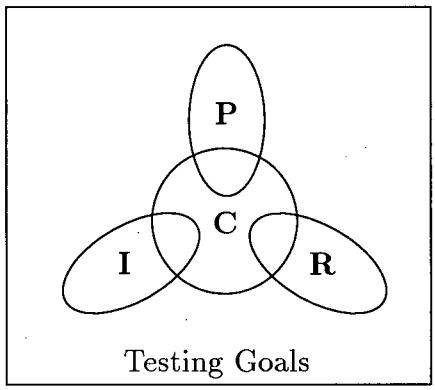


Figure 4.2: The different types of protocol testing and their goals.

process cannot anticipate these errors since the source of some of the errors is *external* to both the tester and the IUT, and is out of their control. This is to be expected since the goal of conformance testing is not to catch this type of error. In fact coordination loss is a kind of problem that should be checked in robustness testing. In other words, there is a relationship between the faults remaining undetected and the type of protocol testing used.

In this case, protocol verification techniques are also not helpful. There is no procedure to check for errors caused by coordination loss since the fault model to be used during the verification process is not known in advance because we cannot precisely identify the possible consequences of coordination loss to the protocol.

Furthermore, most protocol specifications do not deal with all types of errors that may arise from coordination loss. Therefore, this class of errors should be addressed in the design phase.

These facts have some important consequences. Conformance testing is not enough from the point of view of reliability. In fact, only recently have people started to pay more attention to interoperability and performance testing. Ideally, all protocol implementations should be in conformity with their specifications, interoperate in an open system, have adequate performance characteristics, and be reliable. In this chapter we propose a solution to tackle the problem of coordination loss that will improve the reliability of protocol implementations derived from self-stabilizing protocol specifications.

4.2 Important points in the DFT process

1. **Definition of the testing goals to be accomplished.**

   Given the problem of coordination loss, the main goal is to guarantee that a pass verdict is assigned to a test case only if

   (i) a given event e is valid for the current IUT state s according to the testing requirements, and

   (ii) the global predicate that describes the intended protocol behavior is preserved.

   Only the first condition is taken into consideration by the current conformance testing methodology and framework. There is no way of knowing whether the second condition is valid or not by performing conformance testing. Therefore this is a typical problem that should be addressed in the design phase.

2. **Identification of factors that affect the testing goals.**

   The main factor is coordination loss as discussed above and in Section 3.1.2, however there are other factors as discussed in other chapters:

   - In Section 1.3 we have listed five factors that affect the testing of concurrent reactive systems. Among these factors, three affect more directly the testing goals given above: non-reproducible behavior of the system, increased complexity, and lack of a global clock.
   - Software testing is often not based on a fault model (Section 2.2.2). Therefore, when the process of test case generation is performed automatically, it is invariably based solely on the formal specification.
   - Nondeterminism due to concurrency (Section 3.1.1.1) and introduced in the specification (Section 3.1.1.2) — in particular the flexibility factor.

3. **Identification of the factors that can improve the testing goals.**

   Protocols should be designed to deal with coordination loss, and will recover when coordination loss occurs.

Self-stabilization is a paradigm that can be used to design protocols that handle coordination loss. It seems that there is no other general technique to solve this problem.

4. **Definition of the process to achieve the desired testing goals.**
   Create a self-stabilization version of the protocol. This is the main subject of this chapter and will be discussed in Sections 4.4 to 4.6.

4.3 An overview of self-stabilization

In 1974 Dijkstra introduced the concept of self-stabilization [46]. Informally, a distributed system is called self-stabilizing iff it will converge automatically to a safe state in a finite number of steps regardless of the initial state. Of course the meaning of safe and unsafe[2] states depend on the specification of the system.

Self-stabilization makes the initialization of the system irrelevant. Therefore, if we are concerned about fault-tolerant issues, the property of self-stabilization guarantees the system to recover from an unsafe state caused by some perturbations to its normal operation. Some typical situations that might cause a coordination loss in a distributed system are:

- *Inconsistent initialization*—individual processes may be started up in states that are inconsistent with one another.
- *Transmission errors*—messages sent by a sender process may be lost, corrupted, reordered or delivered much later. In this situation the sender's state is no longer consistent with that of the receiver.
- *Reconfiguration errors*—systems may be reconfigured on-the-fly (e.g., addition or deletion of processes, nodes and links) and the new configuration may present an inconsistent view among the processes.
- *Mode change*—it is common to allow a system to operate in different modes depending on different factors such as number of users and load. Due to the distributed nature of the system, the processes may execute in different modes for some time. Eventually,

[2]Also called legal or legitimate, and illegal or illegitimate states respectively.

all processes should converge to the same mode. If this is not the case, the state of the system has become unsafe.

- *Software failure*—if a piece of software (e.g., process or application) becomes temporarily unavailable, its local state may become inconsistent with that of the others when it resumes normal operation.
- *Hardware failure*—the same situation may happen to a piece of hardware such as memory or processor.

Self-stabilization is one of the principles of well-formed protocols as described in Section 3.2.2. The property of self-stabilization provides a built-in safeguard against events that might corrupt the data. In practice, these events are very difficult to catch in the conformance testing process as pointed out in Section 4.1. The interested reader is referred to [154] for a survey on self-stabilization.

4.4 Formal model

In this section we present the communicating extended finite state machine (CEFSM) model used in describing both the communication protocols and the definition of self-stabilization. First we present the nomenclature for identifiers that will be used in this chapter and Chapter 5.

Nomenclature 4.1 (Identifiers) Identifiers can have a subscript and/or a superscript both of which are used to indicate an element in a set. Let $\square_i^x$ represent an identifier. The subscript i refers to the i-th element of $\square$. For example, process P_i. The superscript refers to the x-th element of $\square_i$ in the case $\square_i$ is also a set.

Definition 4.2 (Communicating extended finite state machines) A communicating extended finite state machine is a labeled directed graph where vertices represent states and edges represent transitions. A designated vertex represents the initial state of the machine. Transitions are labeled with the pair $\frac{event}{action}$. An event is the sending of a message, the reception of a message, or an internal event not related to a channel (e.g., a

timeout or a service request from a service user). Messages should be defined in a finite set Σ of message types. The communication between machines is asynchronous. The communicating channels between each pair of machines are not perfect so that messages can be reordered, corrupted or lost. The channels are assumed to have infinite capacity.[3]

Formally, a communicating extended finite state machine P_i $(i = 1 \ldots r)$ is a five-tuple

$$P_i = (S_i, \Sigma_i, \delta_i, \lambda_i, s_i^0),$$

where

- S_i is the set of local states of machine P_i.
- Σ_i is the set of message types that machine P_i can exchange with other machines. This is represented by the sets $\{\Sigma_{i,j}\}$ and $\{\Sigma_{j,i}\}$, respectively. Therefore, $\Sigma_i = \{\Sigma_{i,j}\} \cup \{\Sigma_{j,i}\}$. The set $\{\Sigma_{i,i}\}$ is empty, i.e., machine P_i cannot send or receive messages from itself.
- δ_i is the transition function and is defined as δ_i: $S_i \times \Sigma_i \rightarrow S_i$.
- λ_i is the output function and is defined as λ_i: $S_i \times \Sigma_i \rightarrow \Sigma_i$.
- s_i^0 is the initial state of machine P_i.

In the labeled directed graph that represents a CEFSM, a message preceded by a + sign means that it was received, and a message preceded by a − sign indicates that it was sent.

■

Let P be a protocol specification comprised of processes $P_1, P_2, \ldots, P_r$. Each process is modeled as a CEFSM and they are interconnected by a set of communicating channels $C_1, C_2, \ldots, C_s$. Each process P_i $(i = 1 \ldots r)$ has a finite set of variables $V_i^1, V_i^2, \ldots, V_i^k$. In this model, the concept of a global state plays a fundamental role in the correctness of a communication protocol.

[3]In practice, we can model an infinite buffer using a finite buffer by discarding new incoming messages when the buffer is full.

Definition 4.3 (Global state) The global state of protocol P is defined by the contents of each variable in each process, and the contents of each channel in the protocol. This can be expressed as:

$$\mathcal{S} = (V_1^1, \ldots, V_1^{k_1}) \times (V_2^1, \ldots, V_2^{k_2}) \times \ldots \times (V_r^1, \ldots, V_r^{k_r}) \times C_1 \times C_2 \times \ldots \times C_s \qquad \blacksquare$$

On the other hand, the correctness of a protocol is expressed in terms of the global predicates.

Definition 4.4 (Global predicate) A global predicate represents certain system properties that should hold in all possible protocol computations. The global predicate[4] is defined in terms of global states. ■

Since there is no shared memory in the system where protocol P is running, the local variables of P_i are only updated by commands present in the protocol implementation of P_i. Furthermore, a send command in P_i (**send** $\langle m \rangle$ **to** P_j) appends message $\langle m \rangle$ to the tail of the messages in channel C_{ij}, and a receive command (**rcv** $\langle m \rangle$ **from** j) removes the head message $\langle m \rangle$ from the messages in channel C_{ji}.

Definition 4.5 (Protocol computation) A protocol computation is a sequence of global states. This can be expressed as:

$$\mathcal{C}_P = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_t, \ldots \qquad \blacksquare$$

Since protocols are reactive systems, the computation sequence $\mathcal{C}_P$ is, in general, infinite. In the case $\mathcal{C}_P$ is finite it means that no event/action is enabled in the last state, i.e., all processes are idle.

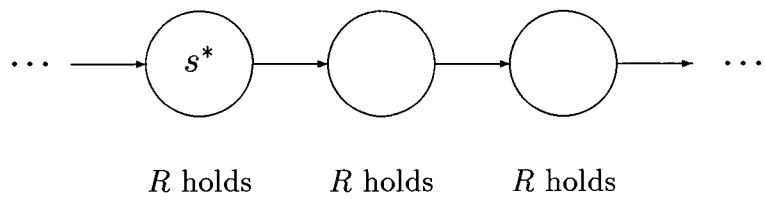
A property of a protocol is defined using global predicates $(R_1, R_2, \ldots)$ that involve global states. Two properties are particularly important for protocols:

- safety properties—defined by closed predicates, and

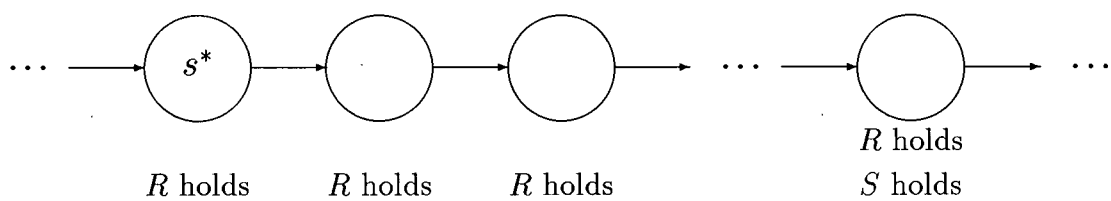
[4]In the remaining of this thesis, the term *predicate* and *global predicate* will be used interchangeably.

- progress properties—defined by a convergence relation over closed predicates.

Definition 4.6 (Closed predicate) A predicate R is called closed iff a state s^* and all other states thereafter that appear in $\mathcal{C}_P$ satisfy R. In this case, s^* and each subsequent state in $\mathcal{C}_P$ is called an R–state. If R and S are two closed predicates of protocol P then R converges to S iff for each possible protocol computation $\mathcal{C}_P$ that starts in an R–state there is a succeeding S-state. This is illustrated in Figures 4.4-(a) and 4.4-(b) respectively. ■



(a) Closed predicate.



(b) Convergence relation.

Figure 4.3: Example of closed predicates.

Note that in a protocol computation $\mathcal{C}_P$ each state is a *true*–state and no protocol state is a *false*–state. Therefore, both *true* and *false* are closed predicates in all protocols. This leads to the definition of self-stabilization.

Definition 4.7 (Self-stabilizing protocol) A protocol P is said to be self-stabilizing iff given any closed predicate R, true converges to R. ■

As mentioned in the definition of global predicate, R should represent a correct behavior of P in all possible protocol computations.

4.5 Design of self-stabilizing protocols

In this section we present a set of novel design principles to incorporate self-stabilization into a protocol specification. These design principles seem to be general enough to apply to a large class of communication protocols as we will see in Section 4.6. To the best of our knowledge there is no work reported in the literature that transforms a given non self-stabilizing protocol into a self-stabilizing protocol in a systematic way. See, for instance, the survey presented by Schneider [154] and the related work in Section 4.8. The main contribution of this chapter is to present a set of design principles to create self-stabilizing protocols in a systematic way.

The design of self-stabilizing protocols can be divided into two steps:

1. Definition of some elements related to the protocol specification.

2. Application of the algorithm to introduce the self-stabilizing features into the protocol specification.

As we will see, these steps define a logical sequence of activities that will end with the stabilization proof of the communication protocol.

In the following each step is discussed in detail.

4.5.1 Elements related to the protocol specification

The elements discussed in this section will be used in Section 4.5.2 when introducing the self-stabilizing features into the communication protocol.

4.5.1.1 Formal model

Each protocol entity should be defined by a communicating extended finite state machine model where the basic elements are states, messages, transitions between states according to some rules defined in the protocol, and the initial state. In particular, the set of messages should be divided into two sub-sets: one contains the messages that can be sent and the other contains the messages that can be received. The communicating channels should also be specified.

The self-stabilizing "features" will be added to the CEFSM model as explained in Section 4.5.2. In the case of the example given in Section 4.6 we will present the code for the protocol that corresponds to the CEFSM model so it is possible to have a more concrete idea of how the self-stabilizing features are introduced in an implementation.

4.5.1.2 Type of communication among entities

Basically, peer entities can communicate one-to-one, one-to-many, many-to-one, and many-to-many. The latter three cases are called group communication. Furthermore, each entity may or may not be autonomous in initiating a communication. A non-autonomous entity can only initiate a communication in response to a message received.

The type of communication among the peer entities should be specified.

4.5.1.3 Timeout actions

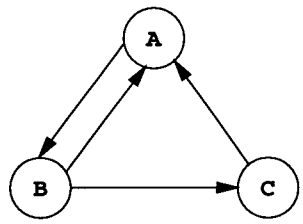
Identify, if any, the timeout actions present in all states indicating the message associated with the timeout.

4.5.2 Algorithm to introduce the self-stabilizing features into the protocol specification

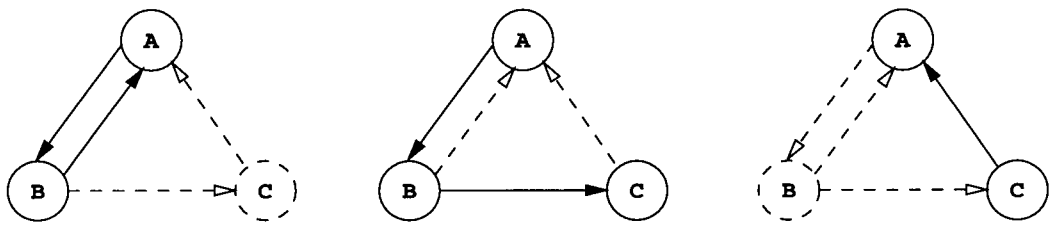
In the algorithm below we use the concepts of phases and paths. Typically a protocol behavior can be partitioned into a number of distinct phases, each one responsible for a specific task. Examples of phases include connection establishment, transfer of data, and orderly termination of the connection. Furthermore, each protocol phase has a set of messages and associated rules for interpreting them. In general, there is only one initial state associated with each protocol phase. Once that initial state is reached, it is assumed that the function performed by the previous phase is completed and the protocol is ready to execute a new function. Of course, this is not valid for the very first phase of a protocol when it starts execution.

It is common to have in each phase two or more distinct paths that reflect the possible outcomes when the phase is executed. A path is defined by the sequence of states traversed in a phase. Some of the paths may return to the initial state of that phase, others may go to the beginning of other phases. In Figure 4.4-(a) a CEFSM with three states that represent

part of a protocol is shown. Suppose there are two phases α and β with initial states A and C, respectively. The paths in this automaton are A–B–A, A–B–C, and C–A as shown by solid lines in Figure 4.4-(b). The first path stays in phase α, the second moves from phase α to β, and the third moves from phase β to α.



(a) Partial CEFSM of a protocol.



(b) Distinct paths (shown by solid lines).

Figure 4.4: Example of different paths in a phase.

The algorithm to introduce the self-stabilizing features into the communication protocol is presented in Figure 4.5. Recall from Definition 4.2 that the CEFSM is a labeled directed graph where vertices and edges represent states and transitions respectively. The algorithm uses the concept of an intermediate vertex, i.e., a vertex in a path except the initial and the last ones.

The algorithm is divided in three parts:

- Lines 2–6: generates all paths in the CEFSM (set *Paths*) and identifies the last transition in each path (set *LTP*).
- Lines 7–9: introduces the lock-step mode principle as explained in Section 4.5.2.1.
- Lines 10–17: introduces timeout actions in each path as explained in Section 4.5.2.2.

Begin of algorithm to introduce self-stabilizing features

Input:

- Directed graph $G = (V, E)$ representing the CEFSM. The set V represents the vertices and the set E the transitions that have the form $\frac{event}{action}$.
- Set $IS = \{v_1, v_2, \ldots, v_p\}$ of vertices that represent the initial state of each phase in the protocol.
- Matrix $T = [\,1..|V|,\ 1..|\Sigma_{output}|\,]$ indicates whether a given vertex has a timeout action associated with the reception of a particular message. The set Σ_{output} represents the set of messages that the machine can send to other machines. The entry $T[i, j]$ is true if vertex v_i has a timeout action associated with message $m_j \in \Sigma_{output}$. Otherwise it is false.

Output:

- Directed graph representing the CEFSM with the self-stabilizing features.

(1) $Paths \leftarrow \{\}$; $LTP \leftarrow \{\}$;

(2) **foreach** vertex $v_i \in IS$ **do**

(3) Find all paths $\pi_i = v_i, v_{i+1}, \ldots, v_l$ in the graph G that start with v_i and finish with v_l, where $v_l \in IS$, such that the only two vertices in the path that are in IS are v_i and v_l.

/* The path π_i can be expressed as a sequence of vertices and edges. Re-writing π_i in terms of states and transitions we have:

$$\pi_i = v_i \frac{event_i}{action_i} v_{i+1} \frac{event_{i+1}}{action_{i+1}} \cdots \frac{event_{l-1}}{action_{l-1}} v_l \qquad */$$

(4) $Paths \leftarrow Paths \cup \pi_i$;

(5) $LTP \leftarrow LTP \cup \frac{event_{l-1}}{action_{l-1}}$;

(6) **od;**

(7) **foreach** pair $\frac{event}{action} \in LTP$ **do**

(8) Generate a new identifier that will be used in all messages in the next path. The generation of the new id should be done as part of the action executed in this transition.

(9) **od;**

(10) **foreach** path $\pi_i \in Paths$ **do**

(11) **foreach** intermediate vertex $v \in \pi_i$ **do**

(12) **if** T[entry associated with v and the output message m that led to v] = **false**

(13) **then** Add a timeout transition to v associated with the message m.

(14) $T[v, m] \leftarrow$ **true;**

(15) **fi;**

(16) **od;**

(17) **od;**

End of algorithm to introduce self-stabilizing features

Figure 4.5: Algorithm to introduce self-stabilization features into a CEFSM.

4.5.2.1 Lock-step mode principle

At each moment in time a process is executing one of the paths in the associated CEFSM. As discussed above, each path represents a possible outcome in the current phase. In order to guarantee coordination among the peer entities, the processes should work in lock-step. Intuitively, this means that the computations among the peer entities should progress together, path-by-path. In fact, this would happen if there were no errors at all in the environment and the protocol is designed with some "nice" properties (e.g., safety and liveness).

To guarantee that the protocol will work in lock-step, each path executed along a computation has associated with it a unique identifier that is monotonically increasing. In this way, each new path executed in each phase by a process can be differentiated from the previous one. The lock-step mode is effectly achieved when all messages exchanged among the peer entities for each path carry the path's identifier. With this mechanism each process knows whether the message is valid for that path or not. If it is not it should be discarded since it does not belong to the current path but to another one that does not exist anymore.

This is an important aspect of our self-stabilization technique. Note that the identifier can be implemented using a timestamp mechanism [99]. This guarantees that the identifiers are monotonically increasing.

The next question is where and when to assign a new identifier for a path. Note that each path $\pi_i \in Paths$ can be represented by the following sequence of vertices (states) and edges (transitions):

$$\pi_i \;=\; v_i \; \frac{event_i}{action_i} \; v_{i+1} \; \frac{event_{i+1}}{action_{i+1}} \; \cdots \; \frac{event_{l-1}}{action_{l-1}} \; v_l.$$

The final state (i.e., vertex) will be the initial state of the current or another phase. Therefore, the right time and place to create a new identifier is when the final state of a phase is reached, i.e., in the transition that leads to v_l. This transition is labeled with the pair $\frac{event_{l-1}}{action_{l-1}}$. Therefore, a new identifier should be generated as part of the action $action_{l-1}$ executed in this transition.

There is also another important aspect related to the generation of identifiers. In Section 4.5.1.2 we classified processes as autonomous or non-autonomous with respect to the

communication among entities. Non-autonomous processes cannot initiate a communication with a peer entity and therefore play a "minor" role in self-stabilization. Typically, their self-stabilization version is obtained by copying the message identifier in a request message to the response message. Autonomous processes are responsible for implementing the lock-step execution mode.

4.5.2.2 Timeout actions

Note that the lock-step execution mode by itself is not sufficient to guarantee self-stabilization. Recall from Section 4.3 that a self-stabilizing protocol will converge automatically to a safe state in a finite number of steps. Therefore, we need a mechanism to guarantee the convergence of the protocol to a safe state. The lock-step principle guarantees that only valid messages are accepted and processed by the protocol. But it does not guarantee that the protocol moves along a given path when it is in an unsafe state. The following theorem guarantees the progress of a protocol execution and thus its convergence to a safe state.

Let s_k be an intermediate state in a path and message m the action associated with the transition from state s_{k-1} to s_k, i.e., $s_{k-1} \xrightarrow{-m} s_k$.

Theorem 4.8 An intermediate state s_k in a path should have a timeout action associated with the output message m that led to that state.

Proof (Contradiction). Suppose that an intermediate state s_k in an autonomous process P does not have a timeout action associated with message m sent to the peer process Q. If process Q is in a state that will not send any reply message then process P can stay in state s_k forever, and thus s_k becomes an unsafe state. Therefore timeout actions should be added for all intermediate states s_k in a path. □

This theorem is based on two assumptions. First, each transition to an intermediate state has the form $\frac{event_{k-1}}{-\langle msg_{k-1}\rangle}$, i.e., an intermediate state is reached by sending a message. Second, once a new phase is reached the function performed by the previous phase has been completed and the protocol is ready to execute a new function. This is the reason for not

including timeout actions in the initial states of each phase. Let us examine this theorem if the assumptions are removed.

If we lift the first assumption given above then an intermediate state s_k can be reached after executing a transition not involving an action related to the communication channel. There are only two possible events that can make the machine move from state s_k to another state: either an event related to the communication channel or a local event such as a request from the service user. In both cases we must have a timeout action associated with this event. Otherwise we have the same situation described in the proof of Theorem 4.8.

If we lift the second assumption above it means that there are overlapping phases in the communication protocol with common intermediate states. This is not a desirable characteristic in the protocol design since two distinct functions are mixed together. In practice, protocols are not designed with overlapping phases. If they do exist then Theorem 4.8 is still valid and must also be applied to a state that is at the same time the end state of a phase and an intermediate state of another phase.

Note that if we keep the second assumption the following theorem holds.

Theorem 4.9 Every initial state of a phase is a safe state.

Proof (Induction). Let set $IS = \{s_1, s_2, \ldots, s_p\}$ be the states that represent the initial state of each phase in the protocol. State s_1 is the initial state of the protocol when it starts to execute. Let $\mathcal{C}_P = s_1, \ldots, s_i, \ldots, s_j, \ldots, s_k, \ldots$ represent the states in a protocol computation of process P where only the states in set IS are shown. Clearly, s_1 is a safe state. Let us assume that state s_j is a safe state and the next state that appears in the computation from the set IS is s_k. If no perturbation occurs in the system between states s_j and s_k, then state s_k is also a safe state according to the assumptions. If a perturbation occurs then the computation will progress until state s_k is eventually reached. This is guaranteed by the timeout actions. Furthermore, only valid messages will be accepted and processed by P because of the lock-step execution mode. Therefore, state s_k is also a safe state. □

A final remark about timeout actions is that they do not apply to non-autonomous processes since they cannot initiate a communication.

4.5.2.3 Complexity of the algorithm

As mentioned earlier, the algorithm has three sequential parts. The first part (lines 2–6) generates all the paths in the graph. This can be done using a breadth-first search algorithm which can be carried out in time $O(v+e)$. The second part (lines 7–9) introduces the lock-step mode principle in each transition of the set LTP. This is clearly bounded by $O(v+e)$ which is the time required to find all the paths. The third part (lines 10–17) introduces the timeout actions along each path. Line 10 is bounded by $O(v+e)$, line 11 by $O(e)$ and lines 12 to 14 are executed in constant time. Thus lines 10 to 17 are bounded by $O(v+e) \times O(e)$ which gives $O(ve+e^2)$. Therefore, the three parts together are bounded by a quadratic function.

If we execute the part associated with the timeout actions when we are generating each path (the first part) we can avoid the cost incurred by the third part and thus the algorithm can be carried out in time $O(v+e)$. The algorithm was presented in three parts for didactical purposes but in a real implementation an optimization like this should be used.

4.6 Self-stabilization: An example

In this section we present a simple protocol for connection management (CM). This protocol is comprised of the connection and disconnection phases. We use this protocol as an example in this chapter because it is present in all connection-oriented protocols and has been widely used. Some of the protocols that use this kind of connection management are the protocols defined for the OSI stack such as the data link, network, transport and session layers, TCP, and protocols for high speed networks such as XTP [162] and NETBLT [37].

4.6.1 Description of the connection management protocol

In the following we describe a connection management protocol where both the connection and disconnection phases use a confirmed service.

Let $Sender_s$ and $Receiver_r$ be the sender process in host s and the receiver process in host r respectively. Informally, the connection management phase works as depicted in Figure 4.6.

$Sender_s$		$Receiver_r$
Sends a $\langle CON.req \rangle$ to $Receiver_r$ ↘		
		↘ $\langle CON.ind \rangle$ received.
		↙ Possible replies: • $\langle CON.resp \rangle$ • $\langle DISC.req \rangle$
• $\langle CON.conf \rangle$ received: Connection established and processes ready for full-duplex data communication. • $\langle DISC.ind \rangle$ received: Sends $\langle DISC.resp \rangle$ ↘		

Figure 4.6: Time-space diagram of the connection management phase.

The disconnection phase works as depicted in Figure 4.7.

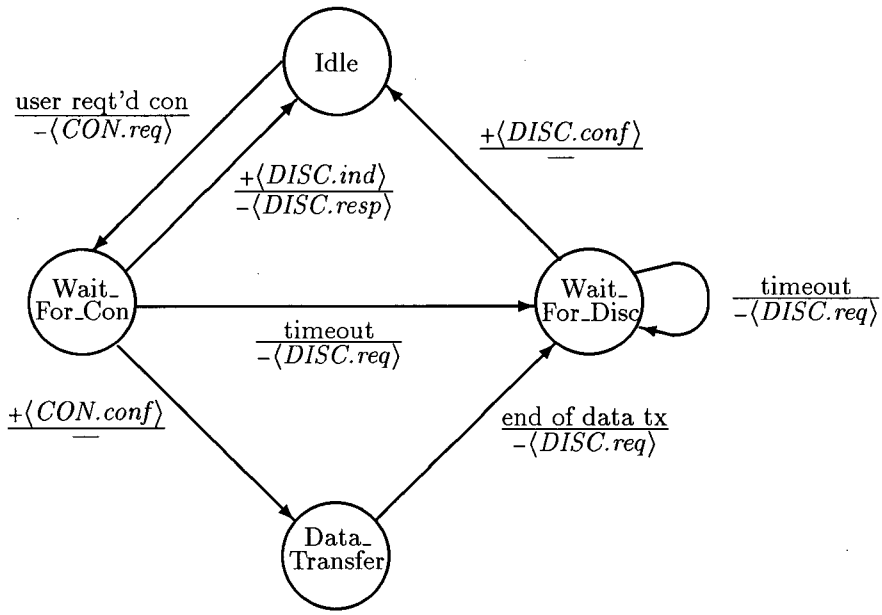
$Sender_s$		$Receiver_r$
Sends a $\langle DISC.req \rangle$ to $Receiver_r$ ↘		
		↘ $\langle DISC.ind \rangle$ received.
		↙ Sends a $\langle DISC.resp \rangle$ to $Sender_s$
$\langle DISC.conf \rangle$ received: Connection is terminated.		

Figure 4.7: Time-space diagram of the disconnection management phase.

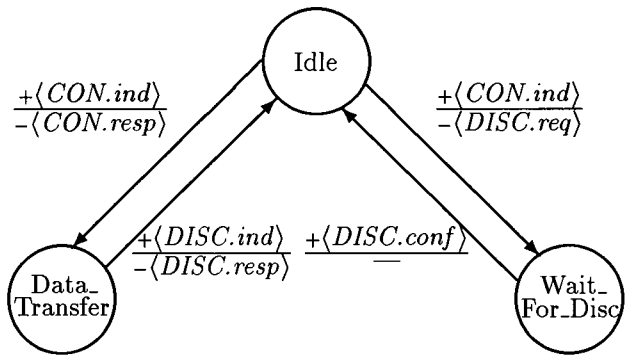
A timeout is also used in the protocol but was not shown in Figure 4.6. If the sender times out, then the message is considered lost and it will be retransmitted. This process may be repeated a finite number of times.

The communicating finite state machines for both processes are shown in Figure 4.8.

The code for processes $Sender_s$ and $Receiver_r$ are given in Figures 4.9 and 4.10 respectively.



(a) Sender process.



(b) Receiver process.

Figure 4.8: Connection management protocol–CEFSM model.

4.6.2 Elements related to the protocol specification

In the following we present the elements related to the protocol specification that will be used in the self-stabilizing version of the protocol as discussed in Section 4.5.1.

```
Begin of Sender_s (original version)
(1)  process Sender_s ≡
(2)  do forever
(3)      (S_s = IDLE ∧ user requests connection) ⟼
(4)          send ⟨CON.req⟩ to r;
(5)          S_s ← WAIT_FOR_CON;
(6)      (S_s = DATA_TRANSFER ∧ data transfer is over) ⟼
(7)          send ⟨DISC.req⟩ to r;
(8)          S_s ← WAIT_FOR_DISC;
(9)      (rcv ⟨m⟩ ∈ Σ_{r,s} from r) ⟼
(10)         do M_r case
(11)             ⟨CON.conf⟩:  S_s ← DATA_TRANSFER;
(12)             ⟨DISC.ind⟩:  send ⟨DISC.resp⟩ to r;
(13)                          S_s ← IDLE;
(14)             ⟨DISC.conf⟩: S_s ← IDLE;
(15)         od;
(16)     (timeout(S_s = WAIT_FOR_CON ∨ WAIT_FOR_DISC)) ⟼
(17)         if S_s = WAIT_FOR_CON then
(18)             send ⟨DISC.req⟩ to r;
(19)             S_s ← WAIT_FOR_DISC;
(20)         else if S_s = WAIT_FOR_DISC then
(21)                 send ⟨DISC.req⟩ to r;
(22)             fi;
(23)         fi;
(24) od;
End of Sender_s (original version)
```

Figure 4.9: Code for $Sender_s$ (original version).

4.6.2.1 Formal model

In the following we identify the set of states and messages valid for both $Sender_s$ and $Receiver_r$, and their communicating channels.

Let S_s indicate the state of $Sender_s$ based on the status of its communicating channel. The states are:

- IDLE: the channel is idle and available for use. No connection is established between $Sender_s$ and $Receiver_r$.
- WAIT_FOR_CON: $Sender_s$ sent a $\langle CON.req \rangle$ (connection request) to $Receiver_r$ and is

Begin of $Receiver_r$ (original version)

```
(1)  process Receiver_r ≡
(2)  do forever
(3)      (rcv ⟨m⟩ ∈ Σ_{s,r} from s) ⟼
(4)          do M_s case
(5)              ⟨CON.ind⟩:  send ⟨CON.resp⟩ to s;
(6)                          S_r ← DATA_TRANSFER;
                      ∨
(7)                          send ⟨DISC.req⟩ to s;
(8)                          S_r ← WAIT_FOR_DISC;
(9)              ⟨DISC.conf⟩: S_r ← IDLE;
(10)             ⟨DISC.ind⟩:  send ⟨DISC.resp⟩ to s;
(11)                         S_s ← IDLE;
(12)         od;
(13) od;
```

End of $Receiver_r$ (original version)

Figure 4.10: Code for $Receiver_r$ (original version).

waiting for a response.

- WAIT_FOR_DISC: $Receiver_r$ sent a $\langle DISC.req \rangle$ (disconnection request) to $Receiver_r$ and is waiting for a response.

- DATA_TRANSFER: a connection between $Sender_s$ and $Receiver_r$ is established and they can start transferring data.

The set of valid messages for $Sender_s$ is $\Sigma_s = \{\langle CON.req \rangle, \langle CON.conf \rangle, \langle DISC.req \rangle, \langle DISC.ind \rangle, \langle DISC.conf \rangle\}$. Let $\Sigma_{r,s}$ indicate the set of valid messages that $Sender_s$ can receive from $Receiver_r$:

- $\langle CON.conf \rangle$: receiver accepted the request made by the sender.

- $\langle DISC.ind \rangle$: receiver rejected the request made by the sender.

- $\langle DISC.conf \rangle$: receiver acknowledged the disconnection request made by the sender.

The set of valid messages that $Sender_s$ can send to $Receiver_r$ is explained in the receiver part since the protocol CM uses a confirmed service.

Let S_r indicate the state of $Receiver_r$. The states are:

- IDLE: the channel is idle and available for use. No connection is established between $Sender_s$ and $Receiver_r$.
- WAIT_FOR_DISC: $Receiver_r$ sent a $\langle DISC.req \rangle$ (disconnection request) to $Sender_s$ and is waiting for a response.
- DATA_TRANSFER: a connection between $Sender_s$ and $Receiver_r$ is established and they can start transferring data.

Note that WAIT_FOR_CON is not a valid state for $Receiver_r$ since upon receipt of a $\langle CON.ind \rangle$ the receiver goes to either the WAIT_FOR_DISC or DATA_TRANSFER state.

The set of valid messages for $Receiver_r$ is $\Sigma_r = \{\langle CON.ind \rangle, \langle CON.resp \rangle, \langle DISC.req \rangle, \langle DISC.ind \rangle, \langle DISC.resp \rangle, \langle DISC.conf \rangle\}$. Let $\Sigma_{s,r}$ indicate the set of valid messages that $Receiver_r$ can receive from $Sender_s$:

- $\langle CON.ind \rangle$: sender requested a connection establishment.
- $\langle DISC.conf \rangle$: receiver rejected the sender's request for a connection establishment and sent a $\langle DISC.req \rangle$ to $Sender_s$.
- $\langle DISC.ind \rangle$: the sender wants to disconnect.

Let C_{sr} indicate the set of messages sent from the sender to the receiver. Similarly, let C_{rs} indicate the set of messages sent from the receiver to the sender.

4.6.2.2 Type of communication among entities

Process $Sender_s$ is autonomous in initiating a communication with $Receiver_r$ whereas the other way around is not possible. $Receiver_r$ only reacts to requests sent by $Sender_s$.

4.6.2.3 Timeout actions

Process $Sender_s$ has timeout actions in the following states:

- WAIT_FOR_CON: associated with a response to the message $\langle CON.req \rangle$.

  In this case $Sender_s$ can receive either a $\langle CON.conf \rangle$ or a $\langle DISC.ind \rangle$.

- WAIT_FOR_DISC: associated with message $\langle DISC.req \rangle$.

  In this case $Sender_s$ can only receive a $\langle DISC.conf \rangle$.

Process $Receiver_r$ does not have any timeout actions since it is not autonomous.

4.6.3 Applying the algorithm to introduce the self-stabilizing features into the protocol specification

The three parts of the algorithm shown in Figure 4.5 are given below.

4.6.3.1 Phases and paths

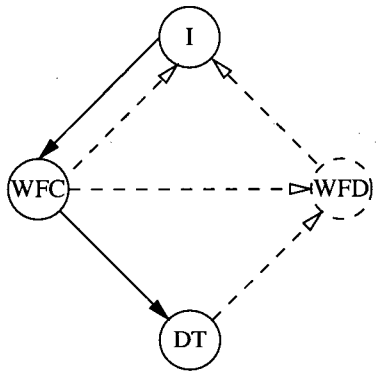
Since process $Receiver_r$ may only react to messages sent by $Sender_s$, this element is relevant to $Sender_s$ only.

The CM protocol as shown in Figure 4.8 is comprised of the connection and disconnection phases and they are treated together in the CEFSM model. The connection phase starts in the state IDLE, and the disconnection phase in DATA_TRANSFER. Therefore, the set of initial states can be defined as $IS = \{$IDLE, DATA_TRANSFER$\}$. Note that the state DATA_TRANSFER is the initial state for the data transfer phase which is not shown in the figure.

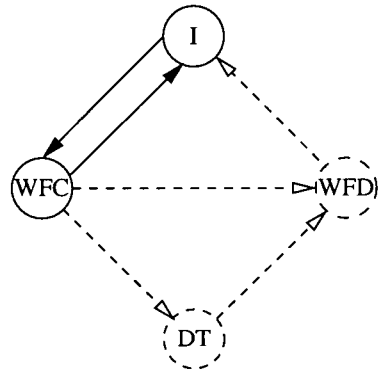
In the case of connection establishment there are three possible outcomes: successful connection, connection rejection, and unsuccessful connection. These possibilities together with the disconnection case define four distinct paths in the graph that represents the CM protocol:

1. Successful connection (Figure 4.11-(a))—$Receiver_r$ accepted a connection request and both entities are ready to start transmitting data.

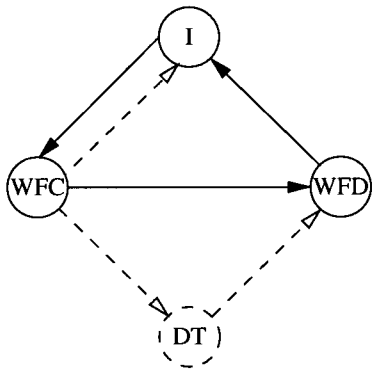
2. Rejection (Figure 4.11-(b))—$Receiver_r$ rejected a connection request.



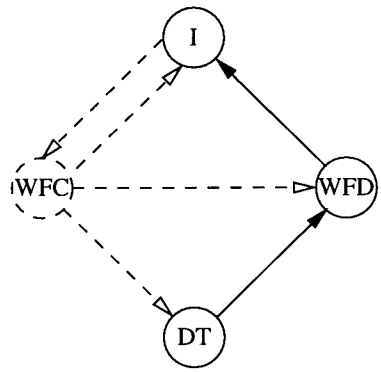
(a) Successful connection.



(b) Rejection.



(c) Unsuccessful connection.



(d) Disconnection.

Figure 4.11: Distinct paths in the connection management protocol.

3. Unsuccessful connection (Figure 4.11-(c))—$Sender_s$ was not able to establish a connection with $Receiver_r$.

4. Disconnection (Figure 4.11-(d))—there was no more data to be transferred to $Receiver_r$ and $Sender_s$ disconnected.

The set *LTP* (last transition of the path) contains three transitions:

1. $\frac{+\langle CON.conf \rangle}{-}$ for the successful connection path.

2. $\frac{+\langle DISC.ind \rangle}{-\langle DISC.resp \rangle}$ for the rejection path.

3. $\frac{+\langle DISC.conf \rangle}{-}$ for the unsuccessful connection and disconnection paths.

4.6.3.2 Lock-step mode

The lock-step mode will be introduced in process $Sender_s$ since it alone can initiate a communication. The variable N_s contains the identifier to be associated with each new path, i.e., N_s is responsible for implementing the timestamp mechanism. The following lines were added to the code of $Sender_s$ as shown in Figure 4.12:

- (10)–(13): to discard any message that does not belong to the current path.
- (17): lock-step mode for the successful connection case.
- (22): lock-step mode for the connection rejection case.
- (26): lock-step mode for the unsuccessful connection and disconnection cases.

All **send** and **rcv** commands in both $Sender_s$ and $Receiver_r$ have a new variable that contains the path id as shown in Figures 4.12 and 4.13, respectively. Note that at any moment in time, a valid message in the channel has an id N, where $N_s - 1 \leq N \leq N_s$. The messages have the id $N = N_s$ when $Sender_s$ is in any state in a path except the last one. When the last state is reached, N_s is incremented and we have $N = N_s - 1$.

At any moment in time, $Sender_s$ is executing *only* one the following actions (given by the line numbers):

- $\ll ConAction \gg$ (3)–(5): user requested a connection.
- $\ll DiscAction \gg$ (6)–(8): user requested a disconnection.
- $\ll RespAction \gg$ (9)–(28): $Sender_s$ received a response.
- $\ll TimeAction \gg$ (29)–(36): a timeout occured.

On the other hand, $Receiver_r$ has just one action to execute:

- $\ll ReplyAction \gg$ (3)–(12): $Receiver_r$ received a request and should reply to it.

These actions are marked along the line numbers in the code.

Figure 4.14 shows the self-stabilizing version of the communicating extended finite state machines for both processes.

Begin of $Sender_s$ (self-stabilizing version)

```
(1)  process Sender_s ≡
(2)  do forever
(3)      (S_s = IDLE ∧ user requests connection) ⟼
(4)          send ⟨CON.req,N_s⟩ to r;
(5)          S_s ← WAIT_FOR_CON;
(6)      (S_s = DATA_TRANSFER ∧ data transfer is over) ⟼
(7)          send ⟨DISC.req,N_s⟩ to r;
(8)          S_s ← WAIT_FOR_DISC;
(9)      (rcv ⟨m ∈ Σ_{r,s}, N⟩ from r) ⟼
(10)         if N ≠ N_s then
(11)             discard message;
(12)             continue;        /* goes to next iteration */
(13)         fi;
(14)         do M_r case
(15)             ⟨CON.conf⟩:  if S_s = WAIT_FOR_CON then
(16)                              S_s ← DATA_TRANSFER;
(17)                              N_s ← N_s + 1;
(18)                          fi;
(19)             ⟨DISC.ind⟩:  if S_s = WAIT_FOR_CON then
(20)                              send ⟨DISC.resp,N_s⟩ to r;
(21)                              S_s ← IDLE;
(22)                              N_s ← N_s + 1;
(23)                          fi;
(24)             ⟨DISC.conf⟩: if S_s = WAIT_FOR_DISC then
(25)                              S_s ← IDLE;
(26)                              N_s ← N_s + 1;
(27)                          fi;
(28)         od;
(29)     (timeout(S_s = WAIT_FOR_CON ∨ WAIT_FOR_DISC)) ⟼
(30)         if S_s = WAIT_FOR_CON then
(31)             send ⟨DISC.req,N_s⟩ to r;
(32)             S_s ← WAIT_FOR_DISC;
(33)         else if S_s = WAIT_FOR_DISC then
(34)                 send ⟨DISC.req,N_s⟩ to r;
(35)             fi;
(36)         fi;
(37) od;
```

End of $Sender_s$ (self-stabilizing version)

Figure 4.12: Code for $Sender_s$ (self-stabilizing version).

```
Begin of Receiver_r (self-stabilizing version)
(1)  process Receiver_r ≡
(2)  do forever
(3)      (rcv ⟨m ∈ Σ_{s,r}, N⟩ from s) ⟼
(4)          do M_s case
(5)              ⟨CON.ind⟩:   send ⟨CON.resp,N⟩ to s;
(6)                           S_r ← DATA_TRANSFER;
                      ∨
(7)                           send ⟨DISC.req,N⟩ to s;
(8)                           S_r ← WAIT_FOR_DISC;
(9)              ⟨DISC.conf⟩: S_r ← IDLE;
(10)             ⟨DISC.ind⟩:  send ⟨DISC.resp,N⟩ to s;
(11)                          S_r ← IDLE;
(12)         od;
(13) od;
End of Receiver_r (self-stabilizing version)
```

Figure 4.13: Code for $Receiver_r$ (self-stabilizing version).

4.6.3.3 Timeout actions

The four paths shown in Figure 4.11 with their respective states are:

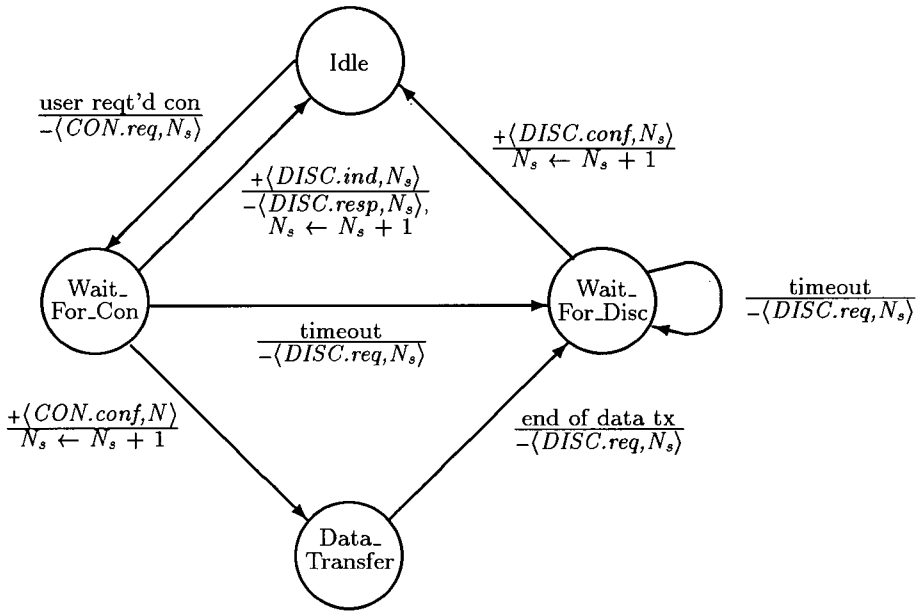
1. Successful connection: I → WFC → DT.

2. Connection rejection: I → WFC → I.

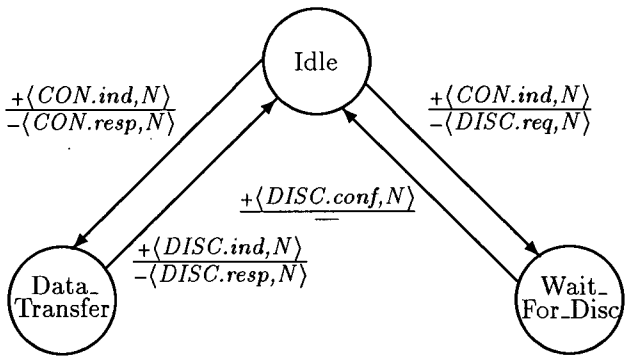
3. Unsuccessful connection: I → WFC → WFD → I.

4. Disconnection: DT → WFD → I.

All four paths have intermediate states and therefore for each one of them we need to check whether it has a timeout action associated with the message that led to that state. The first three paths have the intermediate state WFC (wait for connection) which has a timeout action associated with $\langle CON.req \rangle$. This is the message that led the machine to the state WFC in the three cases. Therefore, we do not need to add any new timeout action. The last two paths above have the intermediate state WFD (wait for disconnection) which has a timeout action associated with $\langle DISC.req \rangle$. This is the message that led the machine to the state WFD in both cases. Therefore no timeout action is needed here as well.



(a) Sender process.



(b) Receiver process.

Figure 4.14: Self-stabilizing version–CEFSM model.

4.6.4 Applying a proof technique to verify the self-stabilization

Gouda and Mutari [71] present a proof technique called convergence stair to show whether a protocol is self-stabilizing for some closed predicate R. The input to the proof technique is the self-stabilizing version P_{SS} of the original protocol P. Therefore, the main problem is to come up with a self-stabilizing version P_{SS} of P such that in the presence of perturbations

the system will converge to a safe state in a finite number of steps where R holds again. This can be accomplished by applying the design principles presented in Section 4.5. In this section we apply the proof technique described in [71] to show that the self-stabilizing version of the connection management protocol is correct.

The convergence stair is defined as a sequence of global predicates $R_1, R_2, \ldots, R_n$ and should satisfy the following three conditions:

(i) *Boundary condition*: $R_1 = true \wedge R_n = R$.

The first predicate is true and holds for an arbitrary initial state. The predicate R is the last one in this sequence and represents the set of legal states of the system.

(ii) *Closure*: Each predicate R_i in the sequence is closed, for $i = 1 \ldots n$.

(iii) *Convergence condition*: Predicate R_i converges to predicate R_{i+1}, for $i = 1 \ldots n-1$.

The convergence stair, as the name suggests, defines a strengthening sequence of global predicates for R.[5] Furthermore, the number of predicates in the convergence stair has a direct impact on the complexity of its proof. The more predicates we have the simpler the proofs will be since the next predicate is "closer" to the current one.

Intuitively the convergence stair says that during a "normal" execution of P, the global predicate R holds at each state in the computation sequence $\mathcal{C}_P$. If there is a perturbation that leads the protocol to an invalid global state, then in a finite number of steps the protocol is guaranteed to converge to a state where R holds again and stay thereafter until a new perturbation occurs.

In the following we present the global predicate and the convergence stair that will be used by the proof technique.

Global predicate

During the connection management phase, if any of the variables that represent local states and channels do not conform to the predicate R then the global state is considered to be

[5]The predicates $R_1, R_2, \ldots, R_n$ represent a strengthening sequence of predicates in the sense that predicate R_i $(i = 2 \ldots n)$ adds some condition to predicate R_{i-1} and thus predicate $R_n = R$ has the strongest conditions.

unsafe.

$R =$

$$(S_s = \text{IDLE} \wedge ((S_r = \text{IDLE} \wedge C_{sr} = C_{rs} = \emptyset) \vee$$
$$(S_r = \text{WAIT\_FOR\_DISC} \wedge C_{sr} = \{\langle DISC.resp \rangle\} \wedge C_{rs} = \emptyset))) \vee$$

$$(S_s = \text{DATA\_TRANSFER} \wedge S_r = \text{DATA\_TRANSFER} \wedge C_{sr} = C_{rs} = \emptyset) \vee$$

$$(S_s = \text{WAIT\_FOR\_CON} \wedge (C_{sr} = C_{rs} = \emptyset \vee$$
$$(S_r = \text{IDLE} \wedge C_{sr} = \{\langle CON.req \rangle\} \wedge C_{rs} = \emptyset) \vee$$
$$(S_r = \text{DATA\_TRANSFER} \wedge C_{sr} = \emptyset \wedge C_{rs} = \{\langle CON.resp \rangle\}) \vee$$
$$(S_r = \text{WAIT\_FOR\_DISC} \wedge ((C_{sr} = \emptyset \wedge C_{rs} = \{\langle DISC.req \rangle\}) \vee$$
$$C_{sr} = C_{rs} = \emptyset)))) \vee$$

$$(S_s = \text{WAIT\_FOR\_DISC} \wedge (C_{sr} = C_{rs} = \emptyset \vee$$
$$(S_r = \text{DATA\_TRANSFER} \wedge C_{sr} = \{\langle DISC.req \rangle\} \wedge C_{rs} = \emptyset) \vee$$
$$(S_r = \text{IDLE} \wedge C_{sr} = \emptyset \wedge C_{rs} = \{\langle DISC.resp \rangle\}) \vee$$
$$(S_r = \text{IDLE} \wedge C_{sr} = \{\langle DISC.req \rangle\} \vee C_{rs} = \emptyset)))$$

The global predicate R represents the cartesian product of the legal states of $Sender_s$ and $Receiver_r$.

The global state for the self-stabilizing version is identical to the one above except that each message is timestamped. The timestamp is contained in the variable N_s.

Convergence stair

Before presenting the convergence stair, some remarks are in order.

For the CM protocol, when the system is in a legal state, there is at most one valid message to be processed either by the sender or by the receiver at any time. If a fault occurred and there are m messages ($m > 1$) in the system then the global state is unsafe. The self-stabilizing version of the protocol will converge to a legal state in a finite number

of steps and the number of messages in the system will go down to one or less.

Consider the following strengthening sequence (R_1, R_2, R_3, R_4) of global predicates of the protocol:

$R_1 = true$

$R_2 = R_1 \wedge (\forall \langle M_s, N \rangle \in C_{sr},\ N \leq N_s)$

$R_3 = R_2 \wedge (\forall \langle M_r, N \rangle \in C_{rs},\ N \leq N_s)$

$R_4 = R_3 \wedge (\,|C_{sr}| + |C_{rs}| = 1\,)$

$R_5 = R$

Global predicate R_1 is the "starting point" of Definition 4.7. Predicate R_2 says that all messages present in the communicating channel from $Sender_s$ to $Receiver_r$ have a sequence number bounded by N_s. Predicate R_3 is equivalent to R_2 but refers to messages from $Receiver_r$ to $Sender_s$. Predicate R_4 says that the number of valid messages in the system at any time is bounded by 1. Predicate R_5 defines the global state of the system.

In this sequence, each predicate strengthens the previous one. Note that the global predicate for the sender (R_2) must come before the predicate for the receiver (R_3) since the sender is the initiating process.

Proof of self-stabilization

Theorem 4.10 The new version of the connection management protocol is self-stabilizing.

Proof. In order to prove that the new version is self-stabilizing we need to verify the convergence stair for R as presented in Section 4.6.4.

(i) *Boundary condition*:

Holds trivially since R_1 and $R_5 = R$.

(ii) *Closure*:

- R_1—trivial.

- R_2—messages in the communicating channel from $Sender_s$ to $Receiver_r$ have a sequence number bounded by N_s. This can be seen if we look where the **send** commands are executed in $Sender_s$:

Action	Line	Command
$\ll ConAction \gg$	(4)	**send** $\langle CON.req, N_s \rangle$ **to** r;
$\ll DiscAction \gg$	(7)	**send** $\langle DISC.req, N_s \rangle$ **to** r;
$\ll RespAction \gg$	(20)	**send** $\langle DISC.resp, N_s \rangle$ **to** r;
	(22)	$N_s \leftarrow N_s + 1$;
$\ll TimeAction \gg$	(31)	**send** $\langle DISC.req, N_s \rangle$ **to** r;
	(34)	**send** $\langle DISC.req, N_s \rangle$ **to** r;

  Lines (4), (7), (31), and (34) of $Sender_s$ have $N = N_s$ when the messages are sent. Also, for the duration of these actions, a new value of N_s is not generated except for the $\ll RespAction \gg$ in $Sender_s$ where N_s is incremented by one in line (22). Therefore all messages present in C_{sr} have $N \leq N_s$. Thus, R_2 holds.

- R_3—messages sent by $Receiver_r$ are responses to messages sent by $Sender_s$. Since R_2 holds and the values of N used by $Receiver_r$ are the same as those in the received message, therefore $N \leq N_s$. Thus, R_3 holds.

- R_4—if R_4 is a closed predicate then any state that follows s^* in $\mathcal{C}_P$ is also an R_4–state. Note that the variable N_s is incremented when $Sender_s$ goes to either state IDLE or DATA_TRANSFER (lock-step mode). In the former case a connection has just been closed or the receiver has rejected a request to open it. In the latter a connection has just been established and both processes are ready to start transmitting data. Both states represent the "starting point" of a new path in the protocol, i.e., a valid response message from $Receiver_r$ should have an id whose value is equal to the current value of N_s in that path.

  Let state s^* be the IDLE state.[6] Predicate R_4 trivially holds at this state since $Sender_s$ did not send any message, and any received message from $Receiver_r$ will be discarded.

[6]Note that in this case we could apply Theorem 4.8 directly. However we will explain why predicate R_4 is closed so we can reason about the self-stabilizing version.

Now, if $|C_{sr}| = 1$ then C_{rs} must be empty since $Receiver_r$ has no timeout action or the right to send a "spontaneous" message. If a message is lost and it is retransmitted, the previous condition still holds. If there is a collision, only one of the received messages from $Receiver_r$ will be considered valid since a path will be completed and N_s will be incremented. Any other message with the previous value of N_s will be discarded.

If $|C_{rs}| = 1$ then C_{sr} must be empty. In this case, $Receiver_r$ is replying to a message sent by $Sender_s$. If C_{sr} is not empty then eventually the first reply corresponding to the sending N_s will complete a path and all messages with the previous value of N_s will be discarded. Thus, R_4 holds.

- R_5—if the system is in a safe state then R_5 holds until a fault occurs when the system will converge to a safe state again.

Therefore, each predicate in the convergence stair is closed.

(iii) *Convergence condition:*

- $R_1 \rightarrow R_2$: When there is one or more messages in channel C_{sr}, all the messages have an identifier N that is less than or equal to N_s, since $Sender_s$ appends the current value of N_s to each message. Eventually, all messages in C_{sr} will be received by $Receiver_r$.
- $R_2 \rightarrow R_3$: Any response from $Receiver_r$ to $Sender_s$ contains the same identifier N as in the received message. If more than one response is received by $Sender_s$ for a given message then it will be discarded since only the first valid reply is considered.
- $R_3 \rightarrow R_4$: This leads to the point where at any moment there is only one valid message in the system. If there were two or more messages this would imply that N_s would have two or more valid values. But N_s in $Sender_s$ has only one value along the time axis so it is not possible to accept two or more response messages for the same requesting message and, therefore, all subsequent messages with the same timestamp (N_s) will be discarded.
- $R_4 \rightarrow R$: All request messages are sent by $Sender_s$ because $Receiver_r$ has no timeout action or the right to send a request. Since R_4 holds, the system will converge to the

global state R.

□

4.6.5 Examples of non self-stabilization and self-stabilization for the CM protocol

In this section we present an example of non self-stabilization for the CM protocol using the original version. This is followed by an example of self-stabilization using the new version.

The proof of non-self-stabilization is shown using a counterexample. We will show that starting at some unsafe state S_0, the protocol goes through other unsafe states $S_1, S_2, \ldots, S_t$ and returns to state S_0. This sequence defines a cycle that is not guaranteed to converge to a legal state in a finite number of steps.

Theorem 4.11 The original connection management protocol is not self-stabilizing.

Proof (counterexample). Assume that the initial state of the system is S_0 which is unsafe. From that state the following sequence of states can be derived from the original specification given in Figures 4.9 and 4.10:

$$S_0 = (S_s = \text{IDLE} \wedge S_r = \text{DATA\_TRANSFER} \wedge$$
$$C_{sr} = \{\langle DISC.req \rangle\} \wedge C_{rs} = \{\langle CON.resp \rangle\})$$

▷ From S_0 to S_1: executing lines 3 to 5 in $Sender_s$

$$S_1 = (S_s = \text{WAIT\_FOR\_CON} \wedge S_r = \text{DATA\_TRANSFER} \wedge$$
$$C_{sr} = \{\langle DISC.req \rangle, \langle CON.req \rangle\} \wedge C_{rs} = \{\langle CON.resp \rangle\})$$

▷ From S_1 to S_2: executing lines 9 to 11 in $Sender_s$

$$S_2 = (S_s = \text{DATA\_TRANSFER} \wedge S_r = \text{DATA\_TRANSFER} \wedge$$

$C_{sr} = \{\langle DISC.req\rangle,\langle CON.req\rangle\} \wedge C_{rs} = \emptyset)$

▷ From S_2 to S_3: executing lines 3–4 and 10–11 in $Receiver_r$

$S_3 = (S_s = \text{DATA\_TRANSFER} \wedge S_r = \text{IDLE} \wedge$
$C_{sr} = \{\langle CON.req\rangle\} \wedge C_{rs} = \{\langle DISC.resp\rangle\})$

▷ From S_3 to S_4: executing lines 6 to 8 in $Sender_s$

$S_4 = (S_s = \text{WAIT\_FOR\_DISC} \wedge S_r = \text{IDLE} \wedge$
$C_{sr} = \{\langle CON.req\rangle,\langle DISC.req\rangle\} \wedge C_{rs} = \{\langle DISC.resp\rangle\})$

▷ From S_4 to S_5: executing lines 3 to 6 in $Receiver_r$

$S_5 = (S_s = \text{WAIT\_FOR\_DISC} \wedge S_r = \text{DATA\_TRANSFER} \wedge$
$C_{sr} = \{\langle DISC.req\rangle\} \wedge C_{rs} = \{\langle DISC.resp\rangle,\langle CON.resp\rangle\})$

▷ From S_5 to S_0: executing lines 9–10 and 14 in $Sender_s$

$S_0 = (S_s = \text{IDLE} \wedge S_r = \text{DATA\_TRANSFER} \wedge$
$C_{sr} = \{\langle DISC.req\rangle\} \wedge C_{rs} = \{\langle CON.resp\rangle\})$

□

Note that according to the OSI conformance testing methodology and framework [87] the sequence of events are valid events since they are valid in each of the states. Therefore they will pass a conformance test despite the fact the implementation is faulty.

Now we present the same example to illustrate self-stabilization.

Suppose that when the following computation sequence started to be executed, the value of N_s was 4.

$S_0 = (S_s = \text{IDLE} \wedge S_r = \text{DATA\_TRANSFER} \wedge$

$C_{sr} = \{\langle DISC.req,\ 3\rangle\} \wedge C_{rs} = \{\langle CON.resp,\ 2\rangle\} \wedge N_s = 4)$

▷ From S_0 to S_1: executing lines 3 to 5 in $Sender_s$

$S_1 = (S_s = \text{WAIT\_FOR\_CON} \wedge S_r = \text{DATA\_TRANSFER} \wedge$
$C_{sr} = \{\langle DISC.req,\ 3\rangle, \langle CON.req,\ 4\rangle\} \wedge C_{rs} = \{\langle CON.resp,\ 2\rangle\} \wedge N_s = 4)$

▷ From S_1 to S_2: executing lines 9 to 12 in $Sender_s$

$S_2 = (S_s = \text{WAIT\_FOR\_CON} \wedge S_r = \text{DATA\_TRANSFER} \wedge$
$C_{sr} = \{\langle DISC.req,\ 3\rangle, \langle CON.req,\ 4\rangle\} \wedge C_{rs} = \emptyset \wedge N_s = 4)$

▷ From S_2 to S_3: executing lines 3–4 and 10–11 in $Receiver_r$

$S_3 = (S_s = \text{WAIT\_FOR\_CON} \wedge S_r = \text{IDLE} \wedge$
$C_{sr} = \{\langle CON.req,\ 4\rangle\} \wedge C_{rs} = \{\langle DISC.resp,\ 3\rangle\} \wedge N_s = 4)$

▷ From S_3 to S_4: executing lines 9 to 12 in $Sender_s$

$S_4 = (S_s = \text{WAIT\_FOR\_CON} \wedge S_r = \text{IDLE} \wedge$
$C_{sr} = \{\langle CON.req,\ 4\rangle\} \wedge C_{rs} = \emptyset \wedge N_s = 4)$

▷ From S_4 to S_5: executing lines 3 to 6 in $Receiver_r$

$S_5 = (S_s = \text{WAIT\_FOR\_CON} \wedge S_r = \text{DATA\_TRANSFER} \wedge$
$C_{sr} = \emptyset \wedge C_{rs} = \{\langle CON.resp,\ 4\rangle\} \wedge N_s = 4)$

▷ From S_5 to S_6: executing lines 9–10 and 15 to 18 in $Sender_s$

$S_6 = (S_s = \text{DATA\_TRANSFER} \wedge S_r = \text{DATA\_TRANSFER} \wedge$
$C_{sr} = \emptyset \wedge C_{rs} = \emptyset \wedge N_s = 5)$

State S_6 satisfies R and thus is a legal state. While this is only one of the possible computation sequences of the protocol all sequences are proved to converge to a safe state

(See Section 4.6.4).

Now, some comments on the self-stabilizing version. In the solution presented, the variable N_s is unbounded. There are two possible solutions to this problem. One may consider that a large variable with 48 or 64 bits is unbounded for practical purposes, or one may use an aperiodic sequence (e.g., a random sequence that is easy to generate) so each path starts with a different sequence number. The latter alternative is called pseudo-stabilization [25]. In practice, these solutions satisfy the needs of most applications.

One particular point related to this protocol is that process $Sender_s$ plays an active role in the self-stabilization process. $Receiver_r$ plays a passive role since it only responds to $Sender_s$. Finally, the following theorem is easily proven.

Theorem 4.12 The number of rounds (in this case receive messages in $Sender_s$) necessary to bring the protocol CM to a safe state when there are m messages in the system is bounded by $\Theta(m)$. □

4.7 A new conformance relation based on the external behavior

This section is related to the theory of testing and is motivated by the design principles described in Section 4.5.

In order to improve the confidence in the implementation under test, it is necessary to define a formal notion of conformance that relates descriptions at different levels of abstraction. In particular, we are interested in two descriptions: specification and implementation. Informally, conformance can be defined as follows:

Definition 4.13 (Conformance relation) Protocol description P_1 conforms to protocol description P_2 (expressed as P_1 **conf** P_2) iff for all possible environments E in which P_1 and P_2 can run, all behaviors of P_1 in E observed at the external interaction points[7] are possible when P_2 is run in the same environment.

[7]In the OSI context, interaction points (IPs) represent service access points, and points of control and observation.

From the point of view of protocol engineering, this is a strong definition which is difficult to realize. It is not practical because the set of possible environments can be very large and hard to be anticipated.

Despite this, several conformance relations that are environment-independent have been proposed in the literature. Some of them are:

- Observational equivalence between CCS processes [128].
- Correctness between a concurrent program and a formula of temporal logic [74].
- The *satisfy*–relation between a CSP process and a formula of trace logic [77].
- The *conf*–relation and testing equivalence between Lotos processes [20]

As pointed out by Gotzhein [70], these relations do not necessarily imply conformance as given in Definition 4.13. Gotzhein describes some problems that arise when different semantics for interaction points are considered in an environment-independent relation.

If we consider an environment that may cause coordination loss and the specification is not designed to handle coordination loss, then the relations above do not help either. Therefore, if the environment is known in advance, a more specific conformance relation can be given. Since coordination loss is a common problem in practice [71, 80, 100, 150, 162], it is quite reasonable to take it into consideration in defining conformance relations. This leads to the following theorem.

Let S_{NSS} and I_{NSS} be a non self-stabilizing protocol specification and its conforming implementation, respectively. Let S_{SS} and I_{SS} be the corresponding self-stabilization versions, respectively. Let E_{CL} be an environment that exhibits coordination loss.

Theorem 4.14 Given the coordination loss fault model, I_{NSS} is a faulty implementation with respect to environment E_{CL} and I_{SS} is not.

Proof (By contradiction). Suppose I_{NSS} is not a faulty implementation with respect to environment E_{CL}. This implies that if an error occurs due to coordination loss, I_{NSS} will converge from an unsafe state to a safe state in a finite number of steps. But this is not

possible because I_{NSS} is not self-stabilizing. That contradicts our hypothesis that I_{NSS} is not a faulty implementation. □

Figure 4.7 shows the relationships among the four versions of specifications and implementations. Despite the fact that I_{NSS} **conf** S_{NSS}, we can see that I_{NSS} $\neg$**conf** S_{SS} and I_{SS} $\neg$**conf** S_{NSS}.

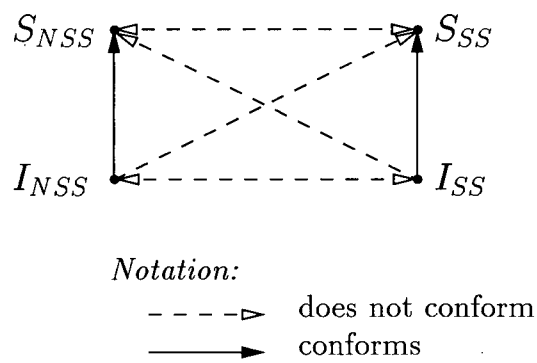


Figure 4.15: Relationships among S_{NSS}, I_{NSS}, S_{SS}, and I_{SS}.

Note that Theorem 4.14 does not say anything directly about the specifications. But since the implementations conform to their specifications, it means that the specification must not accept a faulty behavior related to coordination loss.

This leads us to the following conformance definition that takes into consideration an environment that may cause coordination loss:

Definition 4.15 (Conformance relation conf$_{CL}$) Protocol description P_1 conforms to protocol description P_2 iff for all possible environments that may cause coordination loss (E_{CL}), all behaviors of P_1 in E_{CL} observed at the external interaction points are possible when P_2 is placed in the same environment. Furthermore, any error caused by coordination loss is not valid for P_2. This is expressed as P_1 **conf**$_{CL}$ P_2.

The testing problem discussed in Section 4.1 prompts us to compare the testability of a protocol specification S after embedding its implementation I in environment E_{CL}, and testing I directly. Intuitively, testing through an environment degrades testability. This is another way of interpreting Theorem 4.14. Therefore nothing can be said about the capacity

of the testing process in detecting faulty implementations in an arbitrary environment. This is strongly dependent on S and its implementation I, and on the environment E used for testing.

A natural way of comparing the testability of two implementations in this context is through the relation "more testable" with respect to environment E_{CL}. This is given in the following definition where I_1 and I_2 are implementations of a protocol specification S. Note that E_{CL} defines a fault model.

Definition 4.16 (More testable relation) An implementation I_1 is more testable than implementation I_2 for environment E_{CL} (expressed as $I_1 \sqsubseteq_{E_{CL}} I_2$) if I_1 does not contain any errors that I_2 may have with respect to environment E_{CL} (i.e., the errors that E_{CL} may cause). ■

Clearly, we have $I_{SS} \sqsubseteq_{E_{CL}} I_{NSS}$ for Theorem 4.14 since all errors found in I_{NSS} due to coordination loss will not be present in I_{SS} and therefore I_{SS} is more testable than I_{NSS}. In practice, this relation can be verified if test cases are generated to test all the errors caused by coordination loss, and the tester program is capable of leading the IUT to a state where these errors can be detected. This is difficult to achieve in a conformance testing environment as explained in Section 4.1. Therefore this relation can be re-phrased as follows: implementation I_1 is more *reliable* than implementation I_2 with respect to the errors caused by coordination loss if I_1 does not contain any errors that I_2 may have with respect to this fault model.

4.8 Related work

Gotzhein [70] defines a "compatibility relation" between external interaction points in a system. This relation is used to overcome the problem of defining conformance for an unknown environment. Gotzhein defines several properties that interaction points can have. One of the properties defined prevents duplication, corruption, and creation of interactions (messages). This property can be guaranteed for an interaction point. Unfortunately it cannot be extended to cover different environments where a protocol may be executed. Therefore the problem still remains.

Drira et al. [48, 49] propose a method for analyzing the testability of an implementation I with respect to the verdict of the test execution when I is tested through an environment (see Section 2.6.1.2). They assume that a correct verdict can be assigned to the implementation when I is tested directly when performing conformance testing. Although, they point out that for robustness testing the problem remains. This problem is solved if we apply the design principles described in this chapter.

There are at least two specifications of protocols for high speed networks that try to circumvent the problem of coordination loss by using specific mechanisms. Delta-t [185] uses a time mechanism to guarantee that the lifetime of each packet is bounded and strictly enforced. Sabnani and Netravali [150] propose a transport protocol where protocol entities exchange the full local state periodically independent of changes in the states of the cooperating entities.

Katz and Perry [92] propose a mechanism to create a self-stabilization extension of a distributed program P. The idea is to superimpose onto P a self-stabilizing global monitor that repeatedly performs the following three steps: (i) take snapshots of the global state, (ii) verify whether the snapshots indicate an unsafe global state, and (iii) reset the variables of each process to a safe state when a problem is detected. Of course, each one of these steps "must function correctly no matter what the initial state" is. Note that in this method, the proof of self-stabilization of the system is given by the correctness of the algorithms that implement the method, i.e., the proof is implicit. The processes that execute P have to be modified to send snapshot messages to, and receive reset messages from the global monitor. Our method does not use a global monitor and the modifications introduced into the specification are used to guarantee self-stabilization.

4.9 Conclusions

In this chapter, we have proposed a mechanism for tackling an important class of faults, namely coordination loss, that are very difficult to catch in the testing process. We presented a set of design principles for designing self-stabilizing protocols, and applied these principles to a real protocol. Another example is given in Appendix A.

Note that the design principles proposed in this chapter and design for testability in the

hardware domain share some of the same fundamentals but with different goals. DFT in the hardware domain considers one or more fault models when designing an integrated circuit. The idea is to generate test cases according to these fault models, and then check whether the behavior of the manufactured IC follows the behavior assumed by the fault models (see Section 2.2). Similarly, we consider the coordination loss fault model in the design process in order to avoid errors related to this fault model.

We showed that conformance relations that are environment independent are too generic to deal with errors caused by the environment such as coordination loss. We presented a more realistic conformance relation based on external behavior. An interesting direction that we could follow from here is to define other conformance relations based on other fault models, and incorporate them into the design as well (see Section 6.2).

In designing protocols that are self-stabilizing, we are shifting the task of catching errors due to coordination loss from the testing phase to the design phase where we have a better way of handling this problem. Furthermore, the design principles we have proposed comprise of analysis and synthesis techniques as discussed in Section 2.8.3. This is clearly a *design for testability technique.* As a consequence, we were able to define the "more testable" relation that takes this fact into account, and which reflects the reliability of protocol implementations.

The proposed method of converting a protocol to be self-stabilizing does not present any problem in terms of time or space. The self-stabilizing version should have the same complexity as the original protocol in processing each event. The complexity (or overhead) for a protocol to converge to a safe state depends on the protocol itself. For the protocol studied in this chapter, Theorem 4.12 shows that the complexity is $\Theta(m)$, where m is the number of messages in the system.

It is also interesting to observe that from the point of view of design, the line that separates testability issues from verifiability issues is not very clear. This point will be further discussed in the section related to future work (Section 6.2).

Chapter 5

Dynamic unstable properties and DFT

5.1 Introduction

The validation[1] of global predicates is a fundamental problem in distributed computing that has been used in different contexts such as design, simulation, testing, debugging, and monitoring of distributed programs [6, 64, 155, 156, 171]. A global predicate may either be stable or unstable. Informally, a stable predicate means that once it becomes true during a computation it will remain true after that, such as a system deadlock. An unstable predicate does not have this characteristic. An example is a predicate that relates the length of two queues, each one in a different process. In fact, an unstable predicate may switch from valid to invalid and vice-versa during the execution.

The initial work in the detection of global predicates has concentrated on the validation of stable properties such as distributed termination [47, 60] and deadlock detection [28]. Chandy and Lamport [29] proposed an algorithm to take snapshots in a distributed system that became the basis of other algorithms which check stable properties. The word *snapshot* in this algorithm means the local state of a process P_i in the system. Therefore, when P_i receives a message to take a snapshot it records its local state and "relays the 'take snapshot' message along all of its outgoing channels" [29]. These 'snapshot' messages are used by a global monitor to construct only consistent global states (see Definition 5.4).

Note that a stable property defined in terms of global predicates can be checked by a

[1]The terms checking, detection, and validation will be used interchangeably in this chapter when referring to the validation of desirable behaviors (i.e., predicates or properties) in a protocol.

global monitor which takes snapshots and constructs the consistent global states. If the stable predicate is found to be true in at least one consistent global state constructed from the snapshots taken in the individual processes then it can be inferred that it will remain in that state at the end of the algorithm. If the predicate is false in the global states constructed, then it was also false at the beginning of the algorithm [29].

Unfortunately this approach does not work for unstable predicates which may be true during an execution but not checked, or found to be true in some states but it may have never happened because the global monitor constructs all possible consistent global states. We present an example of this situation in Section 5.2.

The testing process is a run-time activity and we can only hope to detect valid or invalid behaviors in an actual execution of a protocol implementation embedded in a testing environment. In this chapter we focus our attention in the validation of dynamic properties during the testing process and afterwards, during normal execution. Dynamic properties define desirable or undesirable temporal evolutions of the behavior of a communication protocol. We shall present a set of novel design principles that will improve the testing process and the reliability of a protocol implementation by checking desirable behaviors. The goals to be accomplished are summarized as follows:

▷ Protocol testing:

- Mechanism to check global predicates based on local predicates.
- Identification of consistent global states where any type of predicate can be checked.

▷ Protocol execution:

- Obtain information to avoid the problem of state build-up (see Section 1.3).
- Obtain information for use in exception handling.

The rest of this chapter is organized as follows. Section 5.2 discusses the problems faced by a global monitor process when detecting unstable properties. Section 5.3 describes the problem in the context of conformance testing and distributed testing. Section 5.4 discusses the important points that should be followed when developing a new DFT technique. Section 5.5 describes the formal model used in this chapter. Section 5.6 discusses the tasks

involved in the detection of dynamic properties. Section 5.7 describes the design principles related to the testing process, including the algorithm to detect the properties. Section 5.8 describes the design principles related to the execution of the protocol implementation. Section 5.9 discusses the relationship between test case generation and automatic generation of properties. Section 5.10 discusses the related work. Finally, Section 5.11 presents the conclusions for this chapter.

5.2 Global monitors and detection of unstable properties

In the following we present an example based on [6] that illustrates the problem faced by a global monitor when detecting global predicates.

Suppose we have a distributed system S comprised of a set of processes $P_1, \ldots, P_n$ defined as a set of communicating extended finite state machine (see Definition 4.2). In this system there is a special process P_Φ that is responsible for executing a specific function such as testing, debugging, or monitoring some conditions among the processes in S. The conditions can change dynamically with time and are expressed in terms of global predicates. There are two problems that can occur when process P_Φ is checking these predicates. First, the condition to be validated may be true only for some period of time which may not be long enough to be detected. Second, if a global predicate Φ is found to be true by P_Φ we do not know whether Φ ever held during the actual computation or not. These two situations can be exemplified using the distributed computation and the corresponding lattice of consistent global states shown in Figures 5.1–(a) and 5.1–(b) respectively. Nodes in the lattice are labeled with two numbers. The first one represents the event id in process P_1 and the second the event id in process P_2.

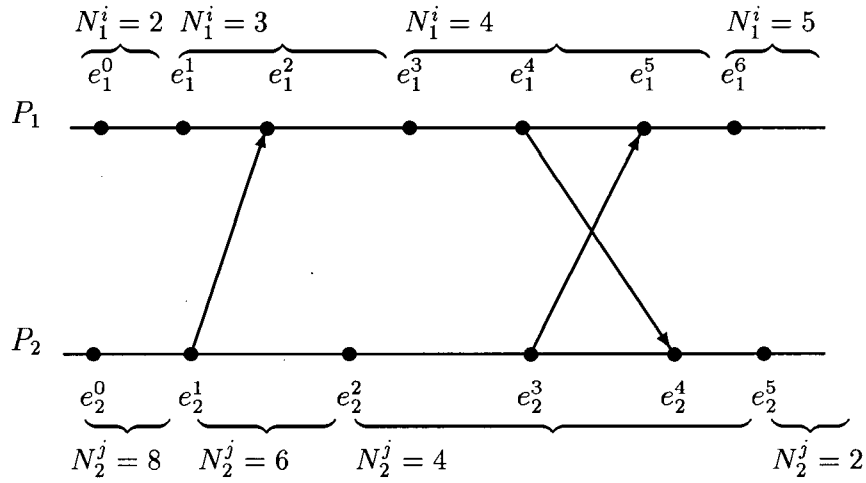
In this example, process P_1 has a local variable N_1^i and process P_2 has a local variable N_2^j. Suppose that process P_Φ is checking the predicate Φ_1: $N_1^i = N_2^j$. (A predicate involving relations among variables in different processes is called a general predicate [6].) If the monitor checks this predicate in any state except the seven inside the dashed rectangle indicated in Figure 5.1–(b) then the condition will not hold. If predicate Φ_2: $N_2^j - N_1^i = 2$ is also checked by P_Φ then there are only two states that satisfy this predicate as shown

in Figure 5.1–(b): states (3,1) and (4,1) in the lattice. Suppose a snapshot was initiated by P_Φ in state (1,1) of the following computation:

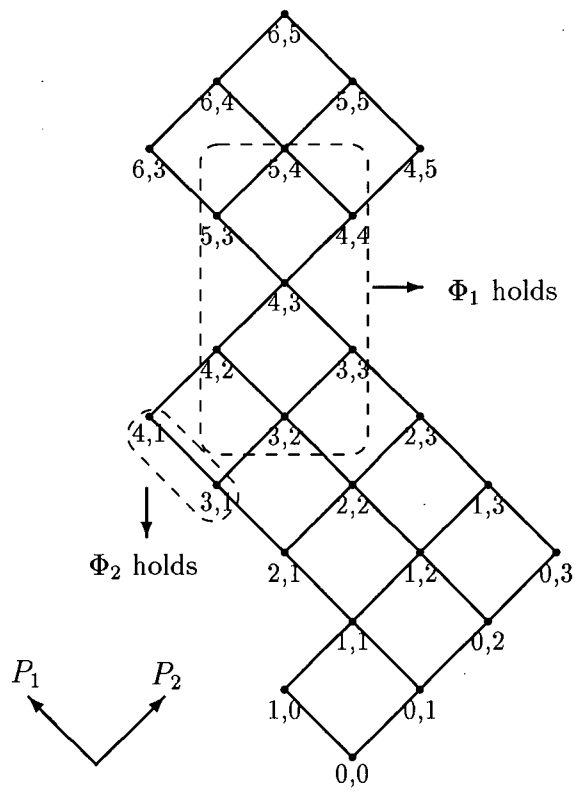
$$\mathcal{C} = (0,0),(0,1),(1,1),(1,2),(2,2),(3,2),(4,2),(4,3),(4,4),(4,5),(5,5),(6,5)$$

In this case, the Chandy-Lamport snapshot algorithm [29] can construct either global state (3,1) or (4,1) since both are reachable from state (1,1). The predicate Φ_2 will be true in these two states although the computation $\mathcal{C}$ did not pass through them.

Cooper and Marzullo [40] proposed a method based on a global monitor that builds the lattice of consistent global states of the distributed computation. The method uses this lattice to analyze *all possible* computations that can occur in the system rather than a *single* computation. Using this approach it is possible to say whether a global predicate Φ is *possibly* or *definitely* true with respect to all computations (also called observations). In the former case a computation possibly satisfies a global predicate Φ iff the lattice contains a global state satisfying Φ. In the latter case a computation definitely satisfies a global predicate Φ iff all possible computations derived from the lattice contain a global state satisfying Φ. In the example of Figure 5.1–(b) we have *Definitely*(Φ_1) and *Possibly*(Φ_2). Unfortunately the size of the lattice can be exponential in the number of events, and therefore, this approach can become infeasible. In Section 5.10 we discuss other approaches proposed in the literature. In this chapter we propose a solution to check unstable properties based on local predicates rather than general predicates. The approach proposed does not use a global monitor and can compute the properties on-line. Furthermore we present a linear algorithm to check these properties (see Section 5.7.3.2).



(a) Time-space diagram of the distributed computation.



(b) Lattice[2] of consistent global states of the distributed computation.

Figure 5.1: Problems faced by a monitor process when detecting global predicates.

[2]This diagram must be read from bottom to top. Therefore the initial state has the label (0,0). A path to the left corresponds to a new event in P_1 and to the right in P_2.

5.3 Conformance testing, distributed testing and problem statement

In this section we discuss the problem of conformance testing and distributed testing of dynamic properties.

Each test method defined in the conformance testing methodology and framework involves three components (see Figure 3.1): (i) a test system responsible for performing the testing process, (ii) the system under test (SUT) that contains the protocol implementation under test (IUT), and (iii) the service-provider responsible for the interconnection between the test system and the SUT.

Clearly there are some similarities between these abstract test methods and the validation of predicates by a global monitor in a distributed system. For instance, part of the test system performs the role of a global monitor. (Recall that all test methods have a lower tester outside of the system under test to exchange protocol data units with the IUT and this is not a function of a global monitor.) The system under test represents the process being monitored. The goal of a test case represents a predicate to be validated, and the predicate validation is made comparing the expected test result with the actual test result. In practice there are several predicates to be validated, each one represented by a different test case.

If a predicate is valid it means that the IUT conforms to the specification with respect to this test case. However, there is an implicit temporal constraint in this statement. The predicate is valid at the moment it was evaluated by the test system. If an unstable predicate is evaluated later it may become invalid.

Also related to this issue is the problem of testing the behavior of the IUT not in isolation but as part of the complete protocol system. Conformance testing can be seen as "half-part" testing in the sense that only one of the parts involved in the system is tested. As mentioned in Section 3.4.1 conformance testing does not guarantee successful interoperability between protocol implementations I_R and I_S of a protocol P. The problem is that the system behavior may not be valid with respect to the specification of P when implementations I_R and I_S interoperate.

A new problem that is receiving more attention lately is related to testing of communication protocols implemented as a set of processes (modules) distributed in a networking environment. The number of system behaviors to be tested increases dramatically due to the concurrent execution of the modules. In Section 5.7 we present a new algorithm to validate dynamic properties in this context.

In summary, there are two questions related to the temporal evolution of the protocol behavior. The problem of checking dynamic properties in a single protocol entity as in conformance testing, and for two or more interacting protocol entities as in interoperability testing. The latter is called distributed testing. In this chapter these two questions will be addressed in a uniform way. In fact, the design principles presented in Sections 5.7 and 5.8 can be applied for distributed software testing in general.

5.4 Important points in the DFT process

In the following, we discuss the testing goals of the properties to be checked in the testing phase.

1. **Definition of the testing goals to be accomplished.**

   The main goal is to be able to check dynamic properties of interest that characterize valid or invalid behaviors for the protocol implementation in distributed testing. These properties should be specified by the designer after the design is completed.

   Note that the designer is the person responsible for identifying these properties and incorporating them into the design. For the lower level design, i.e., the implementation, there are also properties that should be checked based on the information provided in the original design and the implementation code itself.

2. **Identification of the factors that affect the testing goals to be accomplished.**

   The following are the major problems in checking dynamic properties in a distributed environment:

   - The large number of system behaviors to be tested due to the concurrent execution of the modules.

- Checking of global predicates based on a monitor process.
- Difficulty in checking properties that are not stable during an execution. There are two scenarios to consider. First, the property may be invalid from the point of view of the local state (see Definition 5.2). Second, the property may be valid locally, but invalid from the point of view of the global predicate (see Definition 4.4). Suppose in the first case that a property stays invalid temporarily and this condition indicates that there is a fault in the system. If this fault is not sensitized to an output, then the error will remain uncovered. Recall from Section 2.4.1 that the fault-failure model is based on a chain of events that relates inputs, faults, data state errors, and failures. In Chapter 4 we looked at a similar problem related to the second case above. In this chapter, the second scenario will be studied considering local states.

3. **Identification of the factors that can improve the testing goals to be accomplished.**

   In Theorem 5.8 we show that general properties can only be checked by identifying all possible global states of the system. This type of predicate is useful for purposes of debugging when a *post-mortem* analysis is performed.

   Here, we want to devise an algorithm that can check dynamic properties without incurring in the cost of generating all possible global states. Furthermore, this algorithm shall be used in the testing process as well as during normal execution of the protocol. Recall that the cost of generating all possible global states can be exponential in the number of states. This algorithm will check dynamic properties based on local predicates without using a global monitor.

4. **Definition of the process to achieve the desired testing goals.**

   The algorithm to check dynamic properties based on local predicates is described in details in Section 5.7.

5.5 Formal model

In this section we present some definitions that will be used in the algorithm described in Section 5.7.

We will continue to use the communicating extended finite state machine (CEFSM) model presented in Section 4.4. But here we are going to look at a protocol computation $\mathcal{C}_P$ as a partially ordered set of events and states, and their corresponding lattice. Definitions 5.1 to 5.5 are related to these concepts. Definitions 5.6 and 5.7 define local and global predicates respectively.

Definition 5.1 (Causal precedence order) A causal precedence order defines a partial order of events in a system P. Let E be the set of all events that can occur in P and let $\prec$ ("happens before") be the binary relation denoting causal precedence between events as defined by Lamport [99]. Therefore, we have:

$$e_i^a \prec e_j^b \stackrel{\text{def}}{=} \begin{cases} \text{(i):} & (i = j) \wedge (b = a + 1) \vee \\ \text{(ii):} & (e_i^a = \textbf{send } \langle m \rangle \textbf{ to } j) \wedge (e_j^b = \textbf{rcv } \langle m \rangle \textbf{ from } i) \vee \\ \text{(iii):} & \exists\, e_k^c : (e_i^a \prec e_k^c) \wedge (e_k^c \prec e_j^b) \end{cases}$$

Condition (i) says that all events that occur in P_i are totally ordered. Condition (ii) says that if we consider a message $\langle m \rangle$ then the sending event precedes its receiving event. And condition (iii) says that it is possible to define chains of related events based on causality, i.e., relation $\prec$ is transitive. ■

A protocol computation can be represented by a partially ordered set,[3] or poset for short, based on set E, i.e., $\mathcal{C}_P = (E, \prec)$. Graphically, we can represent a distributed computation using the space-time diagram, as shown in Figure 5.1–(a). In this figure, time advances from left to right, and each line represents a distinct process.

If we analyze this space-time diagram we can realize that process P_i enters in a local state s_i^x after event e_i^x happens. It is easy to see that there is a duality between events

[3]Since an event cannot happen before itself, the $\prec$ relation is an irreflexive partial ordering or a precedence relation, i.e., antisymmetric and transitive. Precedence relations share many of the properties of partial ordering relations. However, when a physical situation leads to the definition of a precedence relation, it is often expedient to include the reflexivity property so the terminology and results in connection with partial ordering relations can be applied [111].

and local states in this computation. Let S be the set of states in the computation $\mathcal{C}_P$. Therefore, we can rewrite the binary relation $\prec$ as follows:

$$s_i^a \prec s_j^b \stackrel{\text{def}}{=} \begin{cases} \text{(i):} & (i = j) \wedge (b = a + 1) \vee \\ \text{(ii):} & (e_i^{a+1} = \textbf{send } \langle m \rangle \textbf{ to } j) \wedge (e_j^b = \textbf{rcv } \langle m \rangle \textbf{ from } i) \vee \\ \text{(iii):} & \exists\, s_k^c : (s_i^a \prec s_k^c) \wedge (s_k^c \prec s_j^b) \end{cases}$$

All three conditions have the same meaning as before but in this case we have states instead of events. Therefore, the protocol computation can be represented by another poset based on the set S, i.e., $\mathcal{C}_P = (S, \prec)$. The same distributed computation can be represented using the space-time diagram, as shown in Figure 5.2.

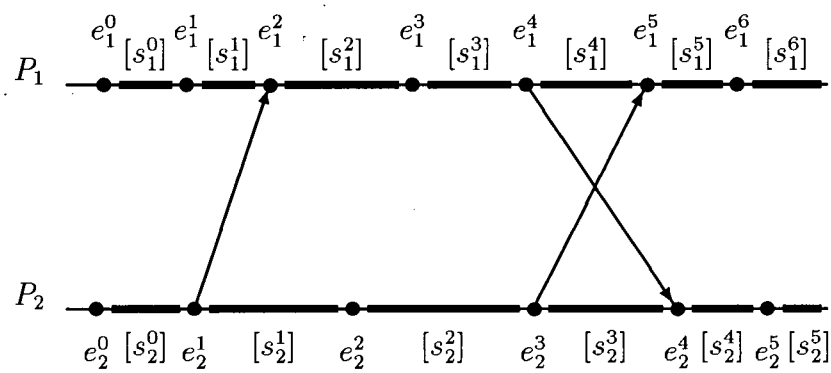


Figure 5.2: Example of a space-time diagram representing a distributed computation based on states.

Definition 5.2 (Local state) The local state γ_i of a process P_i is defined as the contents of each local variable $(V_i^1, \ldots, V_i^{k_i})$ in P_i. ■

In the following, we give a new definition of global states that does not include communication channels as presented in Definition 4.3 followed by the definition of consistent global state. In fact one definition can be transformed into the other if we encode the communication channels as part of each local state.

Definition 5.3 (Global state) A global state is an n-tuple of local states, one for each process P_i, and is represented as follows:

$$\Gamma = (\gamma_1, \ldots, \gamma_n)$$

where γ_i represents the local state of process P_i. ■

Definition 5.4 (Consistent global state) Informally, a global state is consistent if it could occur during an execution and a global clock in the system could be used to label precisely the total order of events. Formally, a global state $\Gamma = (\gamma_1, \ldots, \gamma_n)$ is consistent iff for any pair $(\gamma_i, \gamma_j) \in \Gamma$, then either $\gamma_i \prec \gamma_j$ or $\gamma_j \prec \gamma_i$ for $1 \leq i, j \leq n$ and $i \neq j$, according to the transitive closure of relation $\prec$ between local states. ■

The set of all consistent global states define exactly the states that could have happened in any computation with respect to the events that occurred in each process. Therefore, predicate values are meaningful only if evaluated in a consistent global state.

The set of all consistent global states Γ define a lattice structure $\mathcal{L}$ and its minimal element is the initial global state $\Gamma^0 = (\gamma_1^0, \ldots, \gamma_n^0)$. In the lattice $\mathcal{L}$ there is an edge from a node representing a global state $\Gamma^\alpha = (\gamma_1, \ldots, \gamma_i^a, \ldots, \gamma_n)$ to a node representing $\Gamma^{\text{succ}(\alpha)} = (\gamma_1, \ldots, \gamma_i^{a+1}, \ldots, \gamma_n)$ iff there exists an event e that P_i can execute in its state γ_i^a. Figure 5.1–(b) shows the lattice of the distributed computation given in Figure 5.1–(a). As mentioned before, the 2-tuple is used to represent the global state $\Gamma = (\gamma_1^{a_1}, \gamma_2^{a_2})$.

Definition 5.5 (Sequence of global states) Informally, a sequence of global states represents a computation where the order of each global state in the sequence is given according to a global clock. Thus this sequence represents the serialization of the global states in a particular computation. Formally, a sequence of global states $\Gamma^0, \Gamma^1, \ldots, \Gamma^{\alpha-1}, \Gamma^\alpha, \ldots$ represents a sequence of events $e^1, e^2, \ldots$ that is consistent with the relation $\prec$. In this sequence, global state Γ^α is reached after executing event e^α in global state $\Gamma^{\alpha-1}$. ■

From this definition we can see that there is a duality between sequences of global states and sequences of events. In other words, a sequence of global states define a possible computation of P where the events are implicit, and a sequence of events also define a possible

computation of P where the states are implicit. In either case we have a valid computation of P that starts at the minimum element of lattice $\mathcal{L}$ and goes upwards along one path. Furthermore, this lattice of consistent global states represents all valid observations of P.

Now we present the definitions of local and global predicates that will be used in Section 5.7.

Definition 5.6 (Local predicate) A local predicate of a process P_i is a formula in propositional logic (i.e., a boolean expression) where each term of the formula is a local variable of P_i. The set of local predicates ϕ valid for protocol P can be expressed as:

$$\begin{aligned} \phi &= \{\phi_1^1, \ldots, \phi_1^{p_1}, \phi_2^1, \ldots, \phi_2^{p_2}, \ldots, \phi_n^1, \ldots, \phi_n^{p_n}\} \\ &= \bigcup_{i=1..n} \phi_i \end{aligned}$$

■

Definition 5.7 (Global predicate) A global predicate of protocol P is a formula in propositional logic (i.e., a boolean expression) where each term of the formula is a local predicate of process P_i. A set of global predicates Φ valid for protocol P can be expressed as:

$$\Phi = \{\Phi_1, \Phi_2, \ldots, \Phi_t\}$$

■

5.6 Tasks involved in the detection of unstable dynamic properties

The detection of unstable dynamic properties involves two tasks. The first one defines the ways a property (global predicate) can be expressed, and the second is the design of algorithms to detect such properties. Clearly the rules used for expressing the properties will guide the design of algorithms to detect such properties. For instance, a property can be defined in a general way [40] involving relations among variables in different processes similarly to predicates Φ_1 and Φ_2 in the example of Section 5.2. In this case we need to identify all the consistent global states, which can be represented by a lattice, to determine whether the property definitely or possibly occurred as shown in Theorem 5.8.

Theorem 5.8 Dynamic unstable properties involving relations among variables in different autonomous[4] processes can only be detected by identifying all the consistent global states.

Proof (Contradiction). Without loss of generality let us assume there are two autonomous processes P_1 and P_2 in the system. Suppose we want to check property Φ involving a relation between two variables, each one defined in a different process. If P_1 is capable of detecting property Φ without identifying all the consistent global states it means that P_1 can observe all possible states that P_2 can enter in order to determine precisely whether property Φ holds. But this is not possible because P_1 does not have access to a global clock and P_2 can change its local state. The same argument applies if we consider P_2 as the process capable of detecting the property. Therefore we need to identify all the consistent global states to determine whether the property definitely or possibly occurred.□

In Section 5.10 we compare the different solutions proposed in the literature to represent predicates and their algorithms and the representation used in this thesis with the algorithm proposed in Section 5.7.3.

5.7 Design principles related to testing

This section has four parts. The first part describes how dynamic properties are represented (Section 5.7.1). The second part explains the basic principles used to detect dynamic properties (Section 5.7.2). The third part describes in details the algorithm to check dynamic properties based on local predicates (Section 5.7.3). Finally, the fourth part shows that for a specific type of protocol, namely 3-way handshake protocols, we can compute general properties using this algorithm (Section 5.7.4).

5.7.1 Representation of dynamic properties

Recall from the discussions in Sections 5.1 and 5.3 that we are interested in checking dynamic properties during the testing phase as well as during normal execution of the protocol after the testing process. In this case, the representation of dynamic properties must consider

[4]Recall from Section 4.5.1.2 that autonomous processes can initiate a communication.

two important requirements. First we need to check properties continuously, and second, a dynamic property should describe a valid trace for the protocol specification.

It is well known from automata theory that the languages accepted by finite automata are the languages defined by regular expressions. Furthermore, regular expressions can define infinite languages which, in our case, represent the valid traces for the protocol specification. From Definition 5.7 we can see that a global predicate p is a regular expression which defines a finite trace. If we want to extend the global predicate to represent an infinite language, we must include the operation for concatenation of expressions represented by the symbol $+$ as in p^+.

Definition 5.9 (Dynamic property) A dynamic property is a global predicate expressed in the form given in Definition 5.7 which may contain the operation for concatenation of expressions represented by the symbol $+$. ■

The way a global predicate is expressed is different from equivalence of a regular expression L and a finite automaton A_L in two aspects. First, in the case of language recognition there is only one symbol to be processed at each state of the automaton A_L. In the case of property detection we shall see that we may have a set of valid local predicates in each state of the automaton A_Φ that corresponds to the property Φ. Second, a dynamic property can only use the symbol $+$ for concatenation of expressions and not the symbol $*$. (See the discussion following Theorem 5.12 for not including the symbol $*$.)

From Definition 5.9 it follows that a dynamic property can be expressed by a deterministic finite automaton (DFA). In fact an automaton is a convenient representation for properties since:

- it represents the nature of communication protocols, that is, of reactive systems;
- it is compatible with the communicating extended finite state machines model used in this thesis to represent protocols;
- it allows the protocol behavior to be partitioned, that is paths and phases, where paths represent partial traces.

Definition 5.10 (Deterministic finite automaton) A deterministic finite automaton (DFA) is a directed graph where each transition represents a local predicate, and each automaton state represents an evaluation of the global predicate in some state of the process P_i.

Formally, a deterministic finite automaton A_j $(j = 1 \ldots t)$ is a five-tuple

$$A_j = (Q_j, \Sigma_j, \delta_j, s_j^0, QF_j),$$

where

- Q_j is the set of states of automaton A_j.
- Σ_j is the set of local predicates associated with A_j.
- δ_j is the transition function and is defined as δ_j: $Q_j \times \Sigma_j \rightarrow Q_j$.
- q_i^0 is the initial state of automaton A_j.
- QF_j is the set of accepting states of automaton A_j. ■

Figure 5.3 shows a binary relation between local predicates and global predicates. Let the binary relation μ_i $(i = 1 \ldots n)$ from ϕ_i to Σ be defined by

$$R_{\mu_i} = \{(\phi_i^x, \Sigma_j),\ \phi_i^x \in \phi_i,\ \Sigma_j \in \Sigma \mid \phi_i^x \text{ is a term of } \Phi_j\}.$$

The equivalent matrix representation M_{μ_i} for R_{μ_i} is given in Figure 5.4. The entry $M_{\mu_i}[\phi_i^x, \Sigma_j] = 1$ iff ϕ_i^x is a term in the definition of the global predicate Φ_j. Otherwise it is zero.

Note that if column j of M_{μ_i} has all entries equal to zero it means that predicate Φ_j does not contain any local predicate of ϕ_i and this column can be removed. If line x of M_{μ_i} has all entries equal to zero it means that predicate ϕ_i^x does not appear in any global predicate. The designer should analyze this problem which may be an indication of error.

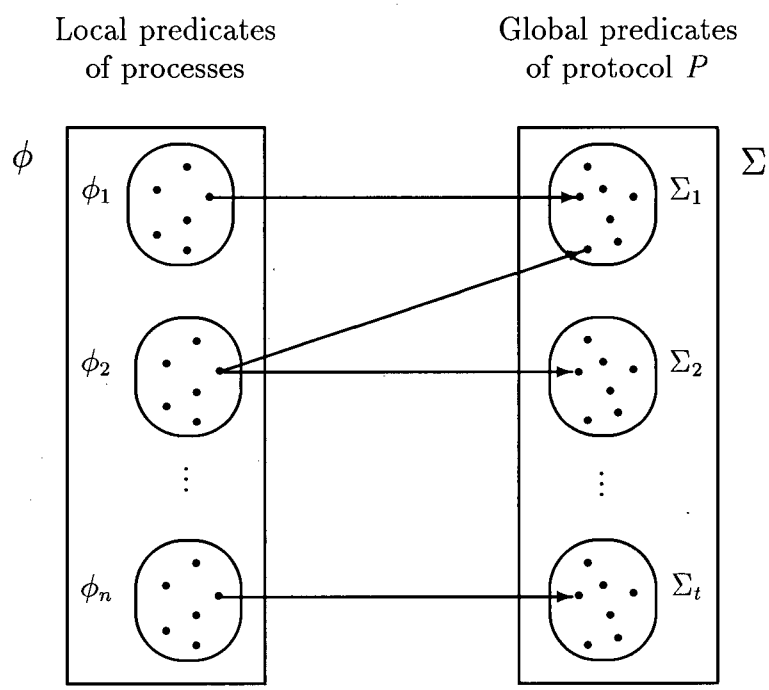


Figure 5.3: Binary relation R_{μ_i} between local and global predicates.

$$M_{\mu_i} = \begin{array}{c|ccccc} & \Sigma_1 & \cdots & \Sigma_j & \cdots & \Sigma_t \\ \hline \phi_i^1 & & & & & \\ \vdots & & & & & \\ \phi_i^x & & & 1 & & \\ \vdots & & & & & \\ \phi_i^{p_i} & & & & & \end{array}$$

Figure 5.4: Matrix representation of binary relation R_{μ_i}.

5.7.2 Detection of dynamic properties

To detect a property we will build the automaton A_j that represents the global predicate Φ_j. The idea of the detection algorithm is to superimpose onto each process P_i $(i = 1 \ldots n)$ that implements protocol P a local checking procedure that repeatedly performs the following steps:

(i) check and determine the valid local predicates in the set ϕ_i when process P_i goes to a new state, and move the automaton A_j to a new state if there are transitions associated with the valid local predicates;

(ii) append the current state of the automaton A_j^i when P_i sends a message to another process; and

(iii) check the local predicates as in (i) when P_i receives a message from P_k and move the automaton A_j to a new state based on the current state and the state received from P_k.

The basic idea of this procedure is to use the automaton A_j to keep track of the protocol behavior. If the behavior exhibited by the protocol is valid, the automaton A_j, which represents property Φ, will eventually reach an accepting state.

Note that this high level algorithm can be applied to any protocol or distributed program in general. In Section 5.7.3 we shall describe this algorithm in detail.

The properties to be checked depend on each protocol and we shall assume that they are part of the input to the algorithm. In Section 5.9 we discuss a related issue which is the problem of test case generation for distributed programs [169] and the automatic generation of properties to be checked.

In the following we apply this high level algorithm to an example. Suppose we have two processes P_1 and P_2 that implement a protocol P such as the ISO Transport Protocol. Furthermore, each process implements a different class of the transport protocol. Given this scenario we would like to make sure, for instance, that the parameters negotiated during the connection establishment remain valid for the entire connection. Let $\phi_1 = \{\phi_1^1, \phi_1^2, \phi_1^3\}$ and $\phi_2 = \{\phi_2^1, \phi_2^2, \phi_2^3, \phi_2^4\}$ represent the set of local requirements for processes P_1 and P_2 respectively. Let us assume that the property that says that "the parameters selected must remain valid for the entire connection" can be expressed as $\Phi_j = (\phi_2^1 \vee \phi_2^2)^+ \wedge \phi_1^1 \wedge \phi_1^2 \wedge (\phi_2^4)^+$. This predicate can be represented by the automaton in Figure 5.5-(a). Figure 5.5-(b) depicts a possible computation for this protocol with the relevant events and messages shown. Next to each event is the set of local predicates that hold at that moment. The current state of the automaton is given inside a circle. Without loss of generality, assume that the protocol P executed the following computation according to the lattice of consistent global states as

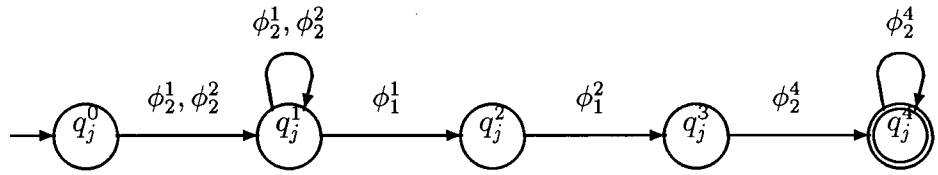
shown in Figure 5.5-(c):

$$(0,0),\ (0,1),\ (1,1),\ (1,2),\ (2,2),\ (2,3),\ (3,3),\ (4,3),\ (4,4)$$

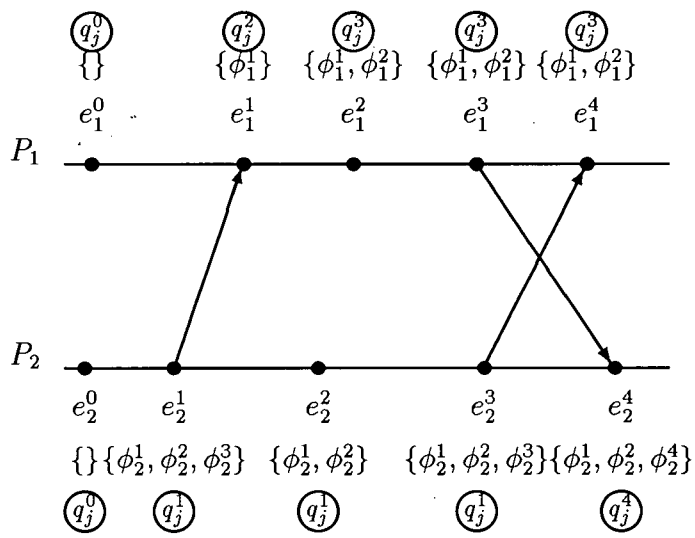
The initial state of the automaton for both processes is q_j^0. When the event e_2^1 occurs the state of the automaton A_j in P_2 goes to q_j^1 after checking the local predicates. At this point, a message $\langle m, q_j^1 \rangle$ is sent to P_1. Process P_1 receives the message and checks its local predicates. At this moment P_1 is in state q_j^0 and P_2 in state q_j^1. Note that the only possible transition in the automaton of P_1 is (q_j^1, ϕ_1^1, q_j^2) and therefore this transition is executed. If the local predicate ϕ_1^1 is not valid when event e_1^1 happens then we must take the automaton to its initial state since we cannot have a valid computation that started at a valid state with an invalid prefix. Then computation proceeds as before. When the event e_2^4 occurs, the state of the automaton A_j in P_2 goes to q_j^4 after checking the local predicates. At this point, a message $\langle m, q_j^3 \rangle$ was received from P_1. Again, the only possible transition in the automaton of P_2 is (q_j^3, ϕ_2^4, q_j^4) and therefore this transition is executed. When the state q_j^4 is reached, the property Φ_j is true in process P_2. Process P_1 will eventually detect this property if the local predicates associated with Φ_j remain true (i.e., ϕ_1^1 and ϕ_1^2) and P_2 sends a message to P_1. If the former condition does not hold it just reflects the temporal behavior of the protocol. If the latter condition does not happen it is because the protocol was not designed to send another message to P_1.

There are two important points that should be noted in the algorithm proposed. First, the properties do not specify interleaved behaviors of the protocol but conditions that should hold with time. This can be seen from the lattice of consistent global states of the distributed computation as shown in Figure 5.5-(c). All possible interleavings can be obtained from the lattice. Independent of which path of computation occurs, the property will hold iff along the path the local predicates of each process are valid when the checking is performed. Second, the checking procedure does not modify the behavior given by the protocol specification, it simply appends the current state of the automaton to each message sent.

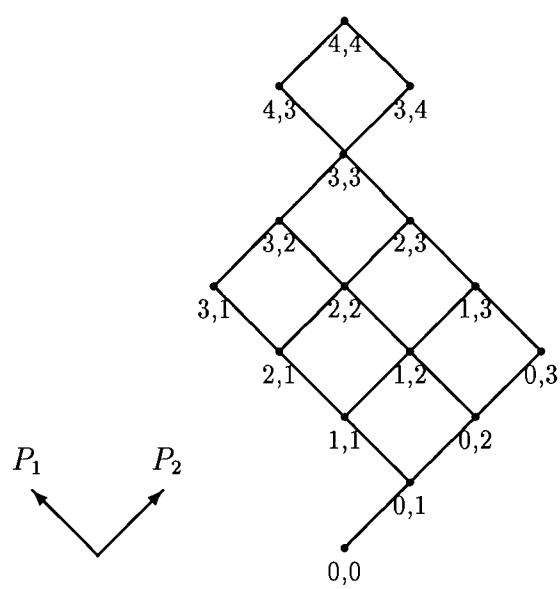
$$\Phi_j = (\phi_2^1 \vee \phi_2^2)^+ \wedge \phi_1^1 \wedge \phi_1^2 \wedge (\phi_2^4)^+$$



(a) Global property and the corresponding DFA.



(b) Time-space diagram of the distributed computation with the valid local predicates at each state.



(c) Lattice of consistent global states of the distributed computation.

Figure 5.5: Checking of properties in a distributed computation.

5.7.3 Algorithm to check dynamic properties based on local predicates

The algorithm to check dynamic properties is given in Figure 5.6 and is divided into several parts as explained below.

5.7.3.1 Data structures and algorithm

▷ Data structures

The data structures used in the algorithm are described in the following:

- *StateOfAutomaton*$[1 \ldots t]$: each entry of this array represents the current state of the Automaton A_j $(j \ldots t)$ in the Process P_i.
- $LP[1 \ldots p_i]$: the x-th entry $(x = 1 \ldots p_i)$ of this array indicates whether the local predicate ϕ_i^x is true at the current state of Process P_i.

  The variable p_i represents the cardinality of the set ϕ_i as given in the Definition 5.6.
- *ValidLP*: set of valid local predicates when process P_i moves to a new state.
- *EndOfTransition*: boolean variable that indicates whether or not there is no more transition to be executed.

▷ Initialization: Lines 1–3

Initializes each automaton to its initial state that is by definition the state q_j^0.

▷ Main part: Lines 4–9

This is the main part of the program that repeatedly checks the properties when the process P_i goes to a new state or appends the array *StateOfAutomaton* to $\langle m \rangle$ when a message is sent.

The part that checks the properties can be seen as a function and is given in lines 10 to 46.

▷ Check each local predicate defined in P_i: Lines 10–12

When process P_i goes to a new state, we need to check each local predicate in order to evaluate the global properties. The truth-value of each local predicate depends on its definition. We give a generic function called *CheckLP* that checks a local predicate.

▷ Validation of global properties: Lines 13–44

In this part we check each property in the set Φ.

▷ Determine the set of valid local predicates in Φ at the current moment: Lines 14–23

An automaton A_j^i can only execute a transition for a local predicate ϕ_i^x iff this local predicate is a term in Φ_j and is valid at the current moment in the process P_i. If the local predicate is a term of Φ_j but is invalid and there is a transition associated with ϕ_i^x at the current state of A_j^i then the automaton A_j^i must be reset. This guarantees the validity of Theorem 5.11.

Theorem 5.11 An execution (trace) that is validated according to the dynamic property Φ does not contain an invalid prefix (sequence of invalid local predicates) after its initial state q_j^0.

Proof. By definition, the initial state q_j^0 is a valid state for the automaton A_j. If A_j changes its state from q_j^0 to q_j^1, it is because there is a local predicate ϕ_i^x valid at state q_j^0 and a transition $(q_j^0,\ \phi_i^x,\ q_j^1)$ exists. If the local predicate ϕ_i^x is invalid then it is not possible to execute the transition. Suppose the current state of automaton A_j is q_j' and it is possible to associate a sequence of valid transitions with the path that led the automaton to this state. Now suppose that $(q_j',\ \phi_i^x,\ q_j'')$ is a possible transition at state q_j' but the local predicate ϕ_i^x is invalid so that it is not possible to execute this transition at this time. If we leave the automaton at this state it is possible this predicate may become true later on and we will execute this transition. That means that if the property Φ eventually holds after the state

q'_j later on, Φ will be true but during the computation there was a false condition. To avoid this inconsistency, we reset the automaton to state q^0_j. □

▷ Determine the next state of the automaton when it receives a message: Lines 24–26

At this point the current state of the automaton A_j is given by *StateOfAutomaton*$[j]$ and a new state q^x_j is received. Therefore, one of these states has to be chosen so the process of checking the property can be carried on. Before showing how this state is determined, let us examine the situations in which an automaton A_j can and cannot move.

The automaton in P_i can move if there is a transition $\mathcal{T} = (q'_j, \phi^x_i, q''_j)$ where q'_j is the current state of A^i_j and $\phi^x_i \in$ *ValidLP* (case i). The automaton cannot move when there is a transition $\mathcal{T}$ but $\phi^x_i \notin$ *ValidLP* (case ii), and when there is no transition $\mathcal{T}$ where $\phi^x_i \in \phi_i$ (case iii). If case (ii) occurs it means that the automaton must be reset as shown in Theorem 5.11. In case (iii), the automaton can only move from this state if there is a predicate ϕ^x_k in process P_k $(k = 1 \ldots n)$ that satisfies condition (i). The automaton must move until one of the conditions (ii) or (iii) happens.

Recall from Definition 5.9 that a global property may contain concatenation of expressions represented by the symbol $+$. Therefore state q^0_j has the following characteristics: (a) it cannot be an accepting state for the automaton; (b) there is no transition that ends at q^0_j; and (c) it has just one transition that starts at q^0_j. Condition (a) follows from (b) but it was intentionally given to emphasize the characteristic of this state.

To determine the state of the automaton we need to enumerate the possible combinations in which processes P_1 and P_2 stopped moving (i.e., conditions (ii) or (iii) above). The combinations are:

Combination	P_1	P_2
1	(ii)	(ii)
2	(ii)	(iii)
3	(iii)	(ii)
4	(iii)	(iii)

Condition (ii) means a reset and condition (iii) means that the move depends on the other process.

Suppose that P_1 is the process that receives a message. (For P_2 is equivalent.) The first combination is trivial and the state of the automaton A_j^1 remains unchanged. The fourth combination shows that the next state of A_j^1 must be the state of the automaton A_j^2 received from the process P_2. The second and third combinations are symmetric. Let us analyze the second combination. The automaton A_j^1 is in the state q_j^0 and the automaton A_j^2 is in the state q_j^x. If the state q_j^x in A_j^2 can be reached directly from state q_j^0 (i.e., with transitions involving only local predicates in P_2) then this situation is similar to the fourth combination and the next state of the automaton must be q_j^x. However, if the state q_j^x cannot be reached directly from q_j^0, it means that process P_1 has already contributed to this path, i.e., there is at least one transition with a local predicate of P_1 before reaching state q_j^x. Since there is no transition that ends at q_j^0 (condition (b) above) it means that the automaton A_j^1 was reset and therefore the next state of A_j^1 must be q_j^0 according to Theorem 5.11. This analysis also applies to the third combination.

Theorem 5.12 When a process P_i receives a message with the state of the automaton A_j in P_k the next state of automaton A_j in P_i can be uniquely determined.

Proof. Given above. □

Note that if we allow the concatenation of expressions represented by the symbol $*$ in the definition of global properties, then the conditions (a), (b), and (c) above do not hold anymore and we cannot apply Theorem 5.12 directly as stated. This does not seem to be an important restriction since if a local property can happen zero or more times, it is probably meaningful if it has to happen at least once.

▷ Determine whether the automaton can move: Lines 27–40

At this point we know the set of valid local predicates and the current state of the automaton A_j and would like to determine if the automaton can move to a new state. This must be

done following the conditions (i), (ii), and (iii) described above.

▷ Check whether the current state of the automaton satisfies the global property: Lines 41–43

If the current state of the automaton A_j is an accepting state then the property Φ is valid at the current state of process P_i.

Begin of algorithm to check dynamic properties

Input:
- Set $A = \{A_1, A_2, \ldots\}$ of automata that represents the set of properties $\Phi = \{\Phi_1, \Phi_2, \ldots\}$ to be detected by process P_i.
- Set $\phi_i = \{\phi_i^1, \ldots, \phi_i^{p_1}\}$ that represents the local predicates valid for process P_i.
- Matrix M_{μ_i} that represents the binary relation R_{μ_i}.

Output:
- Validation of each local predicate in ϕ_i.
- Validation of each dynamic property represented by an automaton in the set A.

```
/* Initialization */
foreach automaton A_j ∈ A do
    StateOfAutomaton[j] ← q_j^0;
od;

/* Main part */
do forever
        (when process P_i goes to a new state) ⟼
            Check properties;
        (when process P_i sends a message ⟨m⟩) ⟼
            Appends the array StateOfAutomaton to ⟨m⟩;
od;
```

↪

Figure 5.6: Algorithm to check dynamic properties (Part 1 of 3).

```
/*** Check properties ***/
/* Validation of each local predicate defined in P_i */
foreach local predicate φ_i^x ∈ φ_i do
    LP[φ_i^x] ← CheckLP(φ_i^x);
od;
/* (P1) */

/* Validation of global properties */
foreach automaton A_j ∈ A do

    /* Determine the set of valid local predicates in Φ at the current moment. */
    ValidLP ← {};
    foreach local predicate φ_i^x ∈ φ_i do
        if M_{μ_i}[φ_i^x, Σ_j] = 1 then
            if LP[φ_i^x] = true then
                ValidLP ← ValidLP ∪ {φ_i^x};
            else /* go to the next iteration */
            fi;
        else /* go to the next iteration */
        fi;
    od;
    /* At this point set "ValidLP" contains the local predicates valid at the current
    moment in P_i. */
```

↪

Figure 5.6: Algorithm to check dynamic properties (Part 2 of 3).

```
        /* Determine the next state of the automaton if a message ⟨m, q_j^x⟩ was received
        from P_k (k = 1...n ∧ k ≠ i). */
(24)    if received message ⟨m, q_j^x⟩ then
(25)       StateOfAutomaton[j] ← state according to Theorem 5.12;
(26)    fi;

        /* Determine whether the automaton can move */
(27)    EndOfTransition ← false;        δ'_j ← δ_j;
(28)    while ¬EndOfTransition do
(29)       if (∃ (StateOfAutomaton[j], φ_i^x, q'_j) ∈ δ'_j)
(30)          if φ_i^x ∈ ValidLP then
(31)                if StateOfAutomaton[j] = q'_j then
(32)                      δ'_j ← δ'_j − (q'_j, φ_i^x, q'_j);
(33)                fi;
(34)                StateOfAutomaton[j] ← q'_j;
(35)          else StateOfAutomaton[j] ← q_j^0;
(36)                EndOfTransition ← true;
(37)          fi;
(38)       else EndOfTransition ← true;
(39)       fi;
(40)    od;

        /* Check whether the current state of the automaton satisfies the global prop-
        erty. */
(41)    if StateOfAutomaton[j] ∈ QF_j then
(42)       /* Global predicate Φ_j is valid at this state. */
           /* (P2) */
(43)    fi;
(44) od;
```

End of algorithm to check dynamic properties

Figure 5.6: Algorithm to check dynamic properties (Part 3 of 3).

5.7.3.2 Complexity of the algorithm

We provide an analysis of the complexity of the algorithm to detect one property Φ_j in lines 13 to 44. Let p_i be the cardinality of the set of local predicates ϕ_i and e the cardinality of the set δ_j. The algorithm has five sequential parts. The first part (lines 10–12) validates each local predicate ϕ_i^x. Suppose that each predicate can be checked in constant time. Then the validation of the local predicates can be carried out in time $O(p_i)$. The second part (lines 14–23) determines the set of valid local predicates in Φ_j which is also executed in time $O(p_i)$. The third part (lines 24–26) determines the next state of the automaton when it receives a message. There are four cases to be analyzed and this part is executed in constant time. The fourth part (lines 27–40) moves the automaton A_j^i, if possible. Since this depends on the number of transitions in A_j this is bounded by $O(e)$. The fifth part checks if the current state of the automaton satisfies the global property and this can done in constant time. Therefore, the five parts together are bounded by $O(p_i + e)$.

5.7.3.3 Remarks about the implementation of the algorithm

Each protocol has a set of phases and paths as discussed in Section 4.5.2. If the sets Φ and ϕ are large, we can evaluate each property $\Phi_j \in \Phi$ only in the phase associated with it. This implies that each local predicate $\phi_i^x \in \phi_i$ will be evaluated in the phases that can change according to the specification.

Since the algorithm is superimposed onto each process P_i it may cause the probe effect. Therefore it should be carefully implemented. One may follow the suggestions given by Bentley in his book entitled "Writing Efficient Programs" [11].

5.7.4 Special global states in the computation

Theorem 5.8 shows that a property Φ involving relations among variables in different autonomous processes (i.e., general predicates) can only be detected by building the lattice of consistent global states and then traversing it to determine whether Φ definitely or possibly is true. This problem arises because we are considering asynchronous distributed systems where the processes are autonomous and may execute at different speeds.

In a synchronous distributed system the coordination among processes happens in global

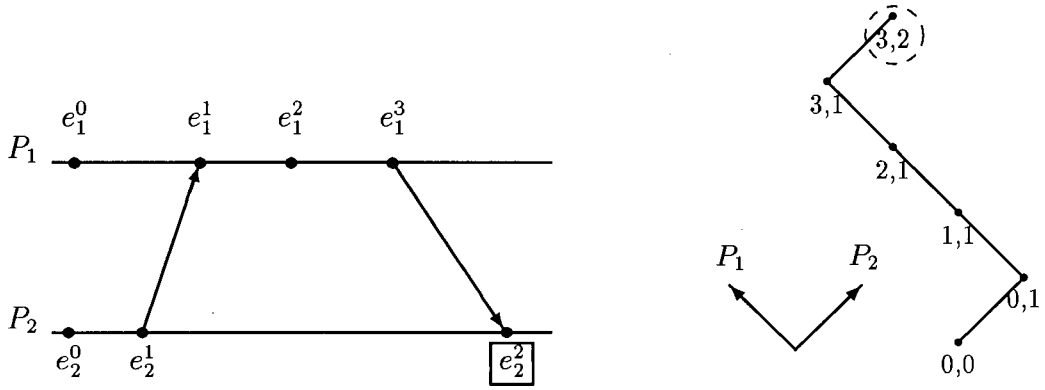
synchronization points. Intuitively this means that the computations of the cooperating processes participating in a synchronization converge to a single global state since all computations have to reach that specific point (state). At that synchronization point, general properties can be evaluated. In asynchronous distributed systems there is a similar situation if there is only one autonomous process P_A participating in a synchronization and P_A is the last process to enter the synchronization state. This is exemplified in Figure 5.7 with two processes.

Suppose process P_2 is autonomous and P_1 is not (as discussed in Section 4.5.1.2). To execute a service in this protocol, P_2 sends a request to P_1 which may reply with a positive or negative response. If the response is positive, P_2 can go to a state $\mathcal{S}_2$ that represents "service accepted." Note that process P_1 is already in a state $\mathcal{S}_1$ that represents "service accepted" by the time P_2 receives the response message. Therefore P_2 was the last process to enter state $\mathcal{S}$ that is our synchronization state. In the executions of Figures 5.7-(a) and 5.7-(b) a positive response is represented by events $\boxed{e_2^2}$, and $\boxed{e_2^4}$ and $\boxed{e_2^5}$ respectively. The global states corresponding to these events in the time-space diagrams are in the lattices. This type of protocol is often called 3-way handshake.

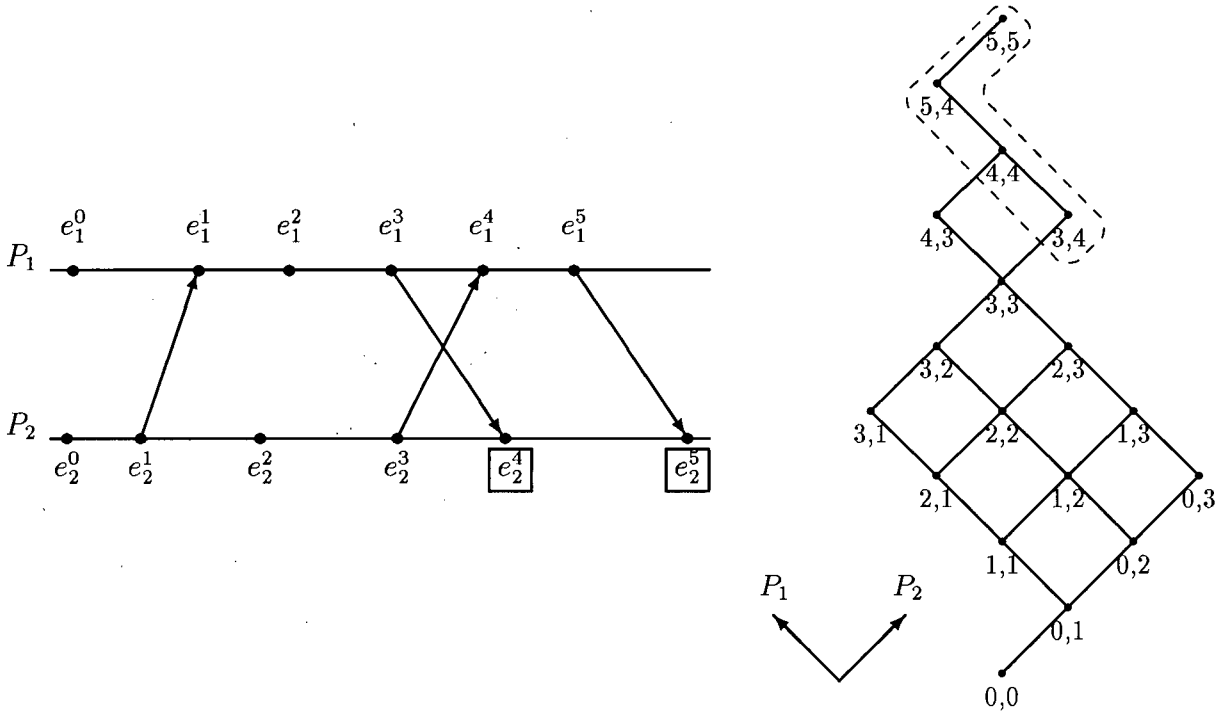
Independent of which computation sequence occurred that contains state $\mathcal{S}_2$ (e.g., $[\ldots,(3,3),(4,3),(4,4),\ldots]$ or $[\ldots,(3,3),(3,4),(4,4),\ldots]$ in Figure 5.7-(b)), we can check general properties at this state since P_1 is in state $\mathcal{S}$ as well. Furthermore, P_1 is not autonomous and therefore will remain at $\mathcal{S}_1$. The same argument can be extended to systems consisting of two or more processes that are non-autonomous and only one autonomous process. This leads to the following theorem.

Theorem 5.13 3-way handshake protocols with two or more non-autonomous processes and only one autonomous process have a global state where general properties can be checked.

Proof. Given above. □



(a) Execution 1.



(b) Execution 2.

Figure 5.7: Special global states in asynchronous distributed systems.

5.8 Design principles related to the execution

The algorithm to detect dynamic properties in Figure 5.6 provides two types of information when process P_i goes to a new state:

1. if each local predicate $\phi_i^x \in \phi_i$ is valid or not, and

2. if each global property $\Phi_j \in \Phi$ is valid or not.

The first information is provided in point (P1) and the second in point (P2) of the algorithm.

Recall from Section 2.4.1 that the fault/failure model [129, 145] relates program inputs, faults, data state errors, and failures. A failure is a manifestation of a fault. Although it may or may not occur when there is a fault in the system.

If the protocol implementation runs in "detection mode" (i.e., with the detection algorithm present) we can use these two pieces of information to tackle the problems of state build-up (see Section 1.3) and exception handling when a fault is detected. Here there is no general solution since these mechanisms depend on the semantics of each protocol. A possible strategy in the case of exception handling is to abort or reinitialize the implementation when "critical" local predicates and properties are not satisfied. This solution is better than allowing an erroneous behavior of the protocol implementation if no action is taken [149]. From the point of view of the other cooperating processes, the action of aborting or reinitializing an implementation can be seen as a physical failure and therefore the protocol should handle it.

5.9 Test case generation and automatic generation of properties

In Section 5.3 we pointed out some similarities between the abstract test methods and the validation of predicates by a global monitor. More specifically, the goal of a test case represents a property to be validated, and the process of evaluating the property is similar to the comparison of the expected test result with the actual test result.

We would like to derive automatically properties from the protocol specification to be used in a distributed testing environment. A possible solution to this problem is to use or adapt test case generation methods for distributed module testing [169] that are not based on constructing a single module. This is an area for future research and not covered in this thesis.

5.10 Related work

The assertional and the operational approaches are commonly used to reason about properties in communication protocols and distributed algorithms.

In the assertional approach, the reachable states are reasoned by means of assertions that are true for all reachable states. This is a static view of the system and represents all possible executions. For instance, in the model checking technique [39] the system is modeled as a finite state graph and properties defined as temporal logic formulas are checked in the reachable states of the graph.

In the operational approach, a concurrent program is analyzed in terms of its behavior, i.e., the events that can occur in the system and the causal precedence relation among these events. This is a dynamic view of the system and represents all possible observations. For example, the method proposed by Cooper and Marzullo [40] described in Section 5.2 can be used to check whether a predicate is possibly or definitely true with respect to all observations. This is clearly a subset of all possible executions that are generated, for instance, in model checking. Generally the assertional and operational approaches are used to check different types of properties. For instance, safety properties in the former case, and dynamic properties in the latter case.

In the following we present the related work with respect to the operational approach in chronological order.

Miller and Choi [122] define linked predicates "that can be ordered by the happened-before relation and are specified by expressions using the $\rightarrow$ operator." They use these predicates in breakpoints in a distributed debugger for halting the system. The halting algorithm is based on the Chandy-Lamport snapshot algorithm [29].

Spezialetti and Kearns [161] consider monotonic events that are similar to stable properties and discuss "conditions which must be met in order for specific assertions to be made about an event or the system state."

Cooper and Marzullo [40] consider general properties which are intractable in practice since they involve building a lattice of consistent global states that can be exponential in the number of events in the system. Furthermore, all possible paths have to be checked.

Hurfin, Plouzeau and Raynal [81] consider unstable nonmonotonic global predicates, called atomic sequences of predicates. These predicates describe global properties by causal composition of local predicates augmented with atomicity constraints that specify forbidden properties.

Garg and Waldecker [64, 184] define properties in terms of boolean expressions using logic operators. The algorithm to detect these properties is implemented using a global monitor that collects information from all processes and evaluates the predicates.

Venkatesan and Dathan [171] also define properties in terms of boolean expressions and give a distributed algorithm to detect these properties. However, the evaluation of properties is performed off-line and they assume that executions of the system are reproducible. Furthermore, they only consider FIFO channels.

The algorithm proposed in this chapter considers properties expressed as boolean expressions with concatenation of expressions represented by the symbol $+$. We give a fully distributed detection algorithm that works on-line and does not modify the protocol specification.

In the case of a centralized monitor P_Φ, each process P_i $(i = 1 \ldots n)$ has to send a message to P_Φ so the property can be checked. Furthermore, if we want to use the information provided by the algorithm during normal execution (see Section 5.8) then the execution in process P_i has to be delayed. The algorithm proposed in this chapter does not have the cost of sending extra messages since it is distributed and does not delay the execution of process P_i.

For an overview of the concepts related to this chapter and some of the approaches proposed in the literature, the interested reader is referred to [156].

5.11 Conclusions

Verification and testing are two complementary techniques and often used during the protocol development cycle.

In the context of protocol testing, most of the properties that characterize valid or invalid behaviors cannot be expressed in terms of stable properties. In fact, these properties are more properly expressed in terms of desirable or undesirable temporal evolutions of the

communication protocol behavior. These properties are inherently unstable since there is no guarantee that they will remain either valid or invalid in a protocol computation.

In this chapter we have presented a new algorithm to detect dynamic unstable properties that can be used in the testing of distributed processes (modules) of a communication protocol. This algorithm provides two types of information that can be used for tackling two problems during program execution: state build-up and exception handling.

The algorithm is based on the observation that given a protocol specification there are multiple valid computations (traces) each of which can be defined by a causal precedence order (Definition 5.1). Dynamic properties are specified by stating conditions (using local predicates) that should hold on all possible computations.

We have also presented a new theorem that can be used to check general properties for a specific type of communication protocol, namely 3-way handshake protocols.

Chapter 6

Conclusions and future work

We conclude this thesis by summarizing the main contributions and presenting suggestions for future work.

6.1 Main contributions

This thesis has three parts:

- The first part describes a framework for reasoning about design for testability of communication protocols.
- The second part presents a set of novel design principles for designing self-stabilizing protocols and two new relations with respect to the testing process.
- The third part describes a set of novel design principles for checking dynamic unstable properties.

In addition, we presented a comprehensive survey of testability and design for testability in the software domain in Chapter 2.

▷ A framework for design for testability of communication protocols

In Chapter 3, we presented a framework that provides a general treatment to the problem of designing communication protocols with testability in mind. Following the protocol engineering life cycle, we have identified and discussed in details issues related to design

for testability in the analysis, design, implementation, and testing phases. The framework presented in Chapter 3 together with the principles discussed in Section 1.4 define the core principles to be used in the design for testability of communication protocols. In Chapters 4 and 5, these principles were applied in the development of new DFT techniques. This framework is the first study that provides this general treatment to the problem of designing communication protocols with testability in mind.

▷ Self-stabilizing protocols and DFT

In Chapter 4, we presented a novel algorithm and the corresponding design principles for tackling an important class of faults, namely coordination loss, that are very difficult to catch in the testing process. These design principles can be applied systematically in the design of self-stabilizing protocols. The viability of these design principles is demonstrated by applying the algorithm to two real protocols.

We showed that conformance relations that are environment independent are too generic to deal with errors caused by the environment such as coordination loss. We presented a more realistic conformance relation based on external behavior, and a "more testable" relation for environments which exhibit coordination loss.

▷ Dynamic unstable properties and DFT

In Chapter 5, we presented a novel algorithm and the corresponding design principles for checking dynamic unstable properties during the testing process. The approach proposed can be used in distributed testing of communication protocols and distributed programs in general. This is a new approach compared to solutions based on global monitors. Furthermore, the information obtained with the algorithm can be used during the normal execution of the protocol implementation to tackle the problems of state build-up and exception handling when a fault is detected.

We also showed that for a specific type of communication protocol, namely 3-way handshake protocols, it is possible to check general properties using the same algorithm.

Finally, a last remark related to protocol design. Gerard Holzmann points out in his

book entitled "Design and Validation of Computer Protocols" [80] that protocol design is still much of an art, but more and more we should strive for applying and defining well-established principles and practices. This thesis has contributed to that goal.

6.2 Suggestions for future work

As mentioned earlier, design for testability of communication protocols has the following characteristics:

- it is a broad and complex research subject;
- it is in continuous evolution since communication protocols with new requirements and constraints are being developed for new applications and environments;
- it is not the kind of problem that once solved the solution is general enough to cover all its aspects.

Therefore, more work is needed in this research area and some are listed below:

- It would be very useful to develop solutions for the several issues discussed in Chapter 3 that affect testing and testability. Some of these issues are refinement of behavioral and structural properties, semantic aspects, expressivity of specification language and implementation language, relation between specification style and implementation, and relating events.
- Recently, Riese and Vijayananda [146] proposed a classification of protocol faults based on the five elements of the protocol design discussed in Section 3.2.1 (i.e., service, assumptions, vocabulary, encoding, and procedure rules). The types of faults identified are based on their experience in performing interoperability testing of different OSI protocols. It would be very useful to design protocols considering the fault models identified in [146] in the same way we did for coordination loss.
- From the point of view of design, the line that separates testability issues from verifiability issues is not very clear. For instance, the behaviors in the design specification to be verified and to be tested are clearly related. Ernberg et al. [57] proposed a specification

method that is claimed to be more amenable to the verification process. In [127], Milne introduced the concept of design for verifiability in the hardware domain, and in [26] Camurati and Prinetto point out that there are several parallels between DFT and DFV (design for verifiability) in the hardware domain. It seems that there is no study in the protocol domain relating design issues to verification and testing. Since the approaches are complementary, it is interesting to investigate this problem.

- It would be very useful to implement the algorithms described in Chapters 4 and 5 in a tool that supports the use of formal description techniques in protocol development [30, 32].

- As pointed out in Section 5.9, we would like to derive automatically properties from the protocol specification to be used in distributed testing. This probably can be done using or adapting test case generation methods for distributed module testing [169] that are not based on a single module. However, this type of test case generation technique is also a recent research topic in protocol testing and more work needs to be done.

- The integration of hardware and software is the most important issue of embedded systems design and it is becoming increasingly important in the new systems being developed. Codesign refers to the integrated design of systems implemented using both hardware and software components. Kalavada and Lee [90] identify three distinct types of hardware/software codesign: (i) joint design of an instruction-set architecture and its program; (ii) synthesis of hardware and/or software from a common specification; and (iii) specification, synthesis, and simulation of heterogeneous systems. It would be very useful to look at testability issues considering the last two types of codesign since the first type is "fundamentally still a chip-level design problem."

Bibliography

[1] Martín Abadi and Leslie Lamport. Composing specifications. SCR Report 66, Systems Research Center, DEC, Palo Alto, CA, USA, October 1990.

[2] ANSI/IEEE Standard 1149.1–1990: Standard Test Access Port and Boundary-Scan Architecture. The Institute of Electrical and Electronic Engineers, 1990.

[3] Noriyasu Arakawa and Terunao Soncoka. A test case generation method for concurrent programs. In *Fourth International Workshop on Protocol Test Systems*, Amsterdam, 1991. North-Holland.

[4] AT&T Technical Journal. Special issue on "Design for X". 69(3), May/June 1990.

[5] Algirdas Avižienis and Chi-Sharn Wu. A comparative assessment of system description methodologies and formal specification languages. Technical Report CSD-900030, Computer Science Department, University of California at Los Angeles, 1990.

[6] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, ACM Press Frontier Series, chapter 4, pages 55–96. ACM Press and Addison-Wesley, second edition, 1993.

[7] Richard Bache and Monika Müllerburg. Measures of testability as a basis for quality assurance. *Software Engineering Journal*, 5(2):86–92, March 1990.

[8] M. Barbeau and B. Sarikaya. A computer-aided design tool for protocol testing. In *Proc. IEEE INFOCOM*, pages 80–86, 1988.

[9] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12, 1969.

[10] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, NY, USA, second edition, 1990.

[11] Jon Louis Bentley. *Writing Efficient Programs.* Prentice-Hall Software Series. Prentice-Hall, Englewood Cliffs, NJ, USA, 1982.

[12] Robert V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.

[13] Manuel Blum. Designing programs to check their work. In *Proceedings of the 1993 International Symposium on Software Testing*, Cambridge, MA, USA, 28–30 Jun 1993.

[14] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 86–97, Seattle, WA, USA, 15–17 May 1989.

[15] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 73–83, Baltimore, MD, USA, 14–16 May 1990.

[16] Gregor von Bochmann. Specifications of a simplified transport protocol using different formal description techniques. *Computer Networks and ISDN Systems*, 18(5):335–377, June 1990.

[17] B. Boehm, J. Brown, H. Kaspar, M. Lipow, J. MacLeod, and M. Merritt. *Characteristics of Software Quality*, volume 1 of *TRW Series of Software Technology*. North-Holland Publishing Company, 1978.

[18] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, January 1984. Reprinted in Richard H. Thayer and Merlin Dorfman (eds.), *Tutorial: System and Software Requirements Engineering*, IEEE Computer Society Press, Washington, DC, USA, 1990.

[19] Barry S. Bosik and M. Ümit Uyar. Finite state machine based formal methods in protocol conformance testing: From theory to implementation. *Computer Networks and ISDN Systems*, 22(1):7–33, 1991.

[20] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS specifications, their implementations and their tests. In Behçet Sarikaya and Gregor von Bochmann, editors, *Protocol Specification, Testing, and Verification, VI*, pages 349–360, Amsterdam, 1986. North-Holland.

[21] Ed Brinksma and Giussepe Scollo. Formal notions of implementation and conformance in LOTOS. Memorandum INF-86-13, University of Twente, The Netherlands, November 1986.

[22] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, March 1984.

[23] F.P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[24] S.P. van de Burgt, J. Kroon, and A.M. Peeters. Testability of formal specifications. In R.J. Linn and M. Ümit Uyar, editors, *Protocol Specification, Testing, and Verification, XII*, Amsterdam, 1992. North-Holland.

[25] James E. Burns, Mohammed G. Gouda, and Raymond E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[26] Paolo Camurati and Paolo Prinetto. Design for verifiability vs. design for testability: Limiting designer's freedom to achieve what? In *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies*, Turin, Italy, 12–14 June 1991.

[27] CCITT Specification and Description Language SDL. Recommendation Z.100. CCITT Blue Book, 1988.

[28] K. Mani Chandy, L.M. Haas, and Jayadev Misra. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[29] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[30] S.T. Chanson, H. Dany, M.C. Kim, Q. Lin, S. Vuong, S. Zhang, L. Zhou, and J. Zhu. The UBC protocol testing environment. In *Sixth International Workshop on Protocol Test Systems*, pages 215–229, 28–30 September 1993. Published as IFIP Transactions C: Communication Systems C–19 1994, North Holland.

[31] S.T. Chanson, A.A.F. Loureiro, and S.T. Vuong. On the design for testability of communication software. In *Proc. of the 24th IEEE International Test Conference*, pages 190–199, Baltimore, MD, USA, 17–21 October 1993.

[32] S.T. Chanson, A.A.F. Loureiro, and S.T. Vuong. On Tools Supporting the Use of Formal Description Techniques in Protocol Development. *Computer Networks and ISDN Systems*, 25(7):723–739, February 1993.

[33] S.T. Chanson, S.T. Vuong, and H. Dany. Ferry clip approach to multi-party testing. In *Third International Workshop on Protocol Test Systems*, Amsterdam, 1990. North-Holland.

[34] S.T. Chanson and Jinsong Zhu. Automatic protocol test suite derivation. In *Proceedings of IEEE INFOCOM*, pages 792–799, Toronto, Ontario, Canada, 12–16 June 1994.

[35] L. Chen and A. Avižienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Dig. Papers FTCS-8: 8th Annu. Int. Conf. Fault Tolerant Comput.*, pages 3–9, Toulose, France, 21–23 June 1978.

[36] Dattakumar M. Chitre and Hsi-Ming Lee. Operation of higher layer data communication protocols over satellite links. *Proceedings of the IEEE*, 78(7):1289–1294, July 1990.

[37] David D. Clark, M. L. Lambert, and L. Zhang. NETBLT: A high throughput transport protocol. Network Information Center RFC 998, SRI International, March 1987.

[38] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM Symposium on Communications Architectures, Protocols and Applications*, pages 200–208, Philadelphia, PA,

USA, 24–27 September 1990. Published as Computer Communication Review, 20(4), September 1990.

[39] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[40] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, Santa Cruz, CA, USA, 20–21 May 1991. Published as ACM SIGPLAN Notices, 26(12), December 1991.

[41] Dan Craigen and Karen Summerskill, editors. *Formal Methods for Trustworthy Computer Systems (FM89)*, Halifax, Nova Scotia, Canada, 23–27 July 1989. Springer-Verlag.

[42] J. Curgus and S.T. Vuong. A metric based theory of test selection and coverage. In *Protocol Specification, Testing, and Verification, XIII*, Amsterdam, 1993. North-Holland.

[43] P.A. Currit, M. Dyer, and H.D. Mills. Certifying the reliability of software. *IEEE Transactions on Software Engineering*, SE-12(1):3–11, January 1986.

[44] Anton T. Dahbura, Krishan K. Sabnani, and M. Ümit Uyar. Formal methods for generating protocol conformance test sequences. *Proceedings of the IEEE*, 78(8):1317–1326, August 1990.

[45] Richard A. DeMillo, W. Michael McCracken, R.J. Martin, and John F. Passafiume. *Software Testing and Evaluation.* The Benjamin/Cummings Publishing Company, Menlo Park, CA, USA, 1987.

[46] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[47] Edsger W. Dijkstra and Carel S. Scholten. Termination detection for diffusing computation. *Information Processing Letters*, 11:217–219, August 1980.

[48] K. Drira, P. Azema, B. Soulas, and A.M. Chemali. A formal assessment of synchronous testability for communicating systems. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 149–156, Pittsburgh, PA, USA, May 1993.

[49] K. Drira, P. Azema, B. Soulas, and A.M. Chemali. Testability of a system through an environment. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. of the 4th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT)*, Orsay, France, 13–17 April 1993. In *Lecture Notes in Computer Science* 668.

[50] Khalil Drira, Pierre Azéma, and François Vernadat. Refusal graphs for conformance tester generation and simplification: A computational framework. In A. Danthine, G. Leduc, and P. Wolper, editors, *Protocol Specification, Testing, and Verification, XIII*, pages E2–1–E2–16, Amsterdam, 1993. North-Holland.

[51] R. Dssouli, K. Karoui, A. Petrenko, and O. Rafiq. Towards testable communication software. In *Eight International Workshop on Protocol Test Systems*, pages 239–255, Evry, France, September 1995.

[52] Rachida Dssouli and Reine Fournier. Communication software testability. In I. Davidson and W. Litwack, editors, *Third International Workshop on Protocol Test Systems*, pages 45–55, Amsterdam, 1990. North-Holland.

[53] Rachida Dssouli and Gregor von Bochmann. Error detection with multiple observers. In Michel Diaz, editor, *Protocol Specification, Testing, and Verification, V*, pages 483–494, Amsterdam, 1985. North-Holland.

[54] Peter van Eijk. *Software Tools for the Specification Language LOTOS.* PhD thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands, 1988.

[55] Peter van Eijk, Harro Kremer, and Marten van Sinderen. On the use of specification styles for automated protocol implementation from LOTOS to C. In Luigi Logrippo, Robert L. Probert, and Hasan Ural, editors, *Protocol Specification, Testing, and Verification, X*, Amsterdam, 1990. North-Holland.

[56] Jan Ellsberger and Finn Kristoffersen. Testability in the context of SDL. In R.J. Linn and M. Ümit Uyar, editors, *Protocol Specification, Testing, and Verification, XII*, Amsterdam, 1992. North-Holland.

[57] Patrik Ernberg, Lars-Åke Fredlund, Hans Hansson, Bengt Jonsson, Fredrik Orava, and Björn Pehrson. Guidelines for specification and verification of communication protocols. Report No. 1, Swedish Institute of Computer Science, Kista, Sweden, 1991. Series: SICS Perspective.

[58] Richard E. Fairley. *Software Engineering Concepts.* McGraw-Hill, 1985.

[59] William B. Frakes, Christopher J. Fox, and Brian A. Nejmeh. *Software Engineering in the UNIX/C Environment.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[60] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.

[61] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, SE-17(6):553–564, June 1991.

[62] Hideo Fujiwara. *Logic Testing and Design for Testability.* MIT Press Series in Computer Systems. The MIT Press, Cambridge, MA, 1985.

[63] Susumu Fujiwara and Gregor von Bochmann. Testing non-deterministic state machines with fault coverage. In *Fourth International Workshop on Protocol Test Systems*, Amsterdam, 1991. North-Holland.

[64] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[65] A. Ghedamsi, R. Dssouli, and G. von Bochmann. Diagnostic tests for single transition faults in non-determinism finite state machines. In *Fifth International Workshop on Protocol Test Systems*, Amsterdam, 1992. North-Holland.

[66] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[67] L.H. Goldstein. Controllability/observability analysis of digital circuits. *IEEE Transactions on Circuits Systems*, CAS-26(9):685–693, September 1979.

[68] Andrzej Goscinski. *Distributed Operating Systems: The Logical Design.* Addison-Wesley, Reading, MA, USA, 1991.

[69] Reinhard Gotzhein. The formal definition of the architectural concept 'interaction point'. In S.T. Vuong, editor, *FORTE '89, 2nd International Conference on Formal Description Techniques*, Amsterdam, 1989. North-Holland.

[70] Reinhart Gotzhein. On conformance in the context of open systems. In *Proc. of the 12th International Conference on Distributed Computing Systems*, pages 236–243, Yokohama, Japan, 9–12 June 1992.

[71] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, C-40(4):448–458, April 1991.

[72] Ajay Gupta. *Synchronization in Distributed Systems.* PhD thesis, The University of Iowa, Department of Computer Science, Iowa City, Iowa, USA, December 1989. Published as Technical Report 90–10, 1990.

[73] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. SRC Report 5, Systems Research Center, DEC, Palo Alto, CA, July 1985.

[74] Brent T. Hailpern. *Verifying concurrent processes using temporal logic*, volume 192 of *Lecture Notes in Computer Science.* Springer-Verlag, 1982.

[75] J.J. Hallenbeck, N. Kanopoulos, and J.R. Cybrynski. The test engineer's assistant: A design environment for testable and diagnosable systems. *IEEE Transactions on Industrial Electronics*, 36(2):278–285, May 1989.

[76] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, January 1985.

[77] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1985.

[78] Daniel Hoffman. Hardware testing and software ICs. In *Proc. of Pacific Northwest Software Quality Conference*, pages 234–244, 1989.

[79] Gerard J. Holzmann. Algorithms for automated protocol validation. *AT&T Technical Journal*, 69(1):32–44, January/February 1990. Special issue on Protocol Testing and Verification.

[80] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall Software Series. Prentice-Hall, 1991.

[81] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, CA, USA, 17–18 June 1993. Published as SIGPLAN Notices, 28(12), December 1993.

[82] B. Huthwaite. Manufacturing comptetiveness and quality by design. In *Proceedings of the 4th International Conference on Product Design for Manufacture and Assembly*, pages 17–25, Newport, RI, USA, 5–6 June 1989. Institute for Competitive Design, Rochester, MN, USA.

[83] IEEE Transactions on Industrial Electronics. Special issue on "Design for Testability". 36(2), May 1989.

[84] ISO IS7498: Information Processing Systems, Open Systems Interconnection. Open Systems Interconnection, Basic Reference Model, 1984.

[85] ISO IS8807: Information Processing Systems, Open Systems Interconnection. LOTOS: A Formal Description Technique for the Temporal Ordering of Observational Behavior, 1989.

[86] ISO IS9074: Information Processing Systems, Open Systems Interconnection. Estelle: A Formal Description Technique Based on an Extended State Transition Model, 1989.

[87] ISO IS9646: Information Processing Systems, Open Systems Interconnection. OSI Conformance Testing Methodology and Framework, 1994. Version 10.6 of 14 March 1994.

[88] Van Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM Symposium on Communications Architectures, Protocols and Applications*, pages 314–329, Stanford, CA, USA, 16–19 August 1988. Published as Computer Communication Review, 18(4), October 1988.

[89] C. L. Jones. A process-integrated approach to defect prevention. *IBM Systems Journal*, 24(2):150–167, 1985.

[90] A. Kalavada and E. A. Lee. Manifestations of heterogeneity in hardware/software codesign. In *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pages 437–438, San Diego, CA, USA, 6–10 June 1994.

[91] Rudolf E. Kalman, Peter L. Falb, and Michael A. Arbib. *Topics in Mathematical System Theory.* McGraw-Hill, New York, NY, USA, 1969.

[92] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 91–101, Quebec City, Quebec, Canada, 22–24 August 1990.

[93] Myung-Chul Kim and Samuel T. Chanson. Design for testability of protocols based on formal specifications. In *Proceedings of the 8th International Workshop on Protocol Test Systems*, Evry, France, September 1995.

[94] H. Kloosterman. Generation of adaptive test cases from nondeterministic finite state models. In *Fifth International Workshop on Protocol Test Systems*, pages 309–320, 28–30 September 1992. Published as IFIP Transactions C: Communication Systems C–11 1993, North Holland.

[95] H. Kloosterman. Test derivation from non-deterministic finite state machines. In *Fifth International Workshop on Protocol Test Systems*, pages 297–308, 28–30 September 1992. Published as IFIP Transactions C: Communication Systems C–11 1993, North Holland.

[96] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[97] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, February 1989.

[98] H. Kremer, J. v.d. Lagemaat, A. Rennoch, and G. Scollo. Protocol design using LOTOS: A critical synthesis of a standardization experience. In Michel Diaz and Roland Groz, editors, *FORTE '92, 5th International Conference on Formal Description Techniques*, Amsterdam, 1992. North-Holland.

[99] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[100] Leslie Lamport. The mutual exclusion problem: Part II – Statements and solutions. *Journal of the ACM*, 33(2):327–348, 1984.

[101] T. van Landegem, M. de Prycker, and F. van den Brande. 2005: A vision of the network of the future. *Electrical Communications*, pages 231–240, 3rd Quarter 1994. Technical journal of Alcatel, France.

[102] Kim G. Larsen. *Context-Dependent Bisimulation Between Processes.* PhD thesis, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1986.

[103] Kim G. Larsen. Ideal specification formalism = expressivity + composionality + decidability + testability + ... In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90: Theories of concurrency–unification and extension*, pages 33–56. Volume 458 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990.

[104] Kim G. Larsen and Bent Thomsen. A modal process logic. In *3rd Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Edinburgh, Scotland, 5–8 July 1988.

[105] Hareton K.N. Leung and Lee White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, Miami, FL, USA, 16–19 October 1989.

[106] Hareton K.N. Leung and Lee White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance*, pages 290–301, November 1990.

[107] H. Paul Lin and Harry E. Stovall, III. Self-synchronizing communication protocols. *IEEE Transactions on Computers*, C-38(5):609–625, May 1989.

[108] Richard J. Linn, Jr. Conformance evaluation methodology and protocol testing. *IEEE Journal on Selected Areas in Communications*, 7(7):1143–1158, September 1989.

[109] Richard J. Linn, Jr. Conformance testing for OSI protocols. *Computer Networks and ISDN Systems*, 18(3):203–219, 1990.

[110] Richard J. Lipton. New directions in testing. In Joan Feigenbaum and Michael Merrit, editors, *Distributed Computing and Cryptography*, pages 191–202. American Mathematical Society, 1989. Published as DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 2, 1991.

[111] C. L. Liu. *Elements of Discrete Mathematics.* McGraw-Hill Computer Science Series. McGraw-Hill, second edition, 1985.

[112] Ming T. Liu. Protocol engineering. In Marshall C. Yovits, editor, *Advances in Computers*, volume 29, pages 79–195. Academic Press, San Diego, CA, USA, 1989.

[113] Gang Luo, Anindya Das, and Gregor von Bochmann. Software testing based on SDL specifications with save. *IEEE Transactions on Software Engineering*, SE-20(1):72–87, January 1994.

[114] Gang Luo, G. von Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, February 1994.

[115] Zohar Manna and Amir Pnueli. On the faithfulness of formal methods. In A. Tarlecki, editor, *16th International Symposium on Mathematical Foundations of Computer Science*, pages 28–42, Kazimierz Dolny, Poland, 9–13 September 1991. In *Lecture Notes in Computer Science* 520.

[116] James Martin and Carma McClure. *Software Maintenance: The Problem and Its Solutions.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1983.

[117] Colin M. Maunder and Rodham E. Tulloss. Testability on TAP. *IEEE Spectrum*, 29(2):34–37, February 1992.

[118] R. G. Mays, C. L. Jones, G. J. Holloway, and D.P. Studinski. Experiences with defect prevention. *IBM Systems Journal*, 29(1):4–32, 1990.

[119] M. McAllister, S.T. Vuong, and J. Alilovic-Curgus. Automated test case selection based on test coverage metrics. In *Fifth International Workshop on Protocol Test Systems*, Amsterdam, 1992. North-Holland.

[120] Edward J. McCluskey. *Logic Design Principles.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1986.

[121] C.E. McDowell and D.P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[122] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, CA, USA, 13–17 June 1988.

[123] Raymond E. Miller. The construction of self-synchronizing finite state protocols. *Distributed Computing*, 2(2):104–112, 1987.

[124] Raymond E. Miller and Sanjoy Paul. Generating conformance test sequences for combined control and data flow of communication protocols. In R.J. Linn and M. Ümit Uyar, editors, *Protocol Specification, Testing, and Verification, XII*, Amsterdam, 1992. North-Holland.

[125] Harlan D. Mills. Zero defect software: Cleanroom engineering. In Marshall C. Yovits, editor, *Advances in Computers*, volume 36, pages 1–41. Academic Press, San Diego, CA, USA, 1993.

[126] Harlan D. Mills, M. Dyer, and R.C. Linger. Cleanroom software engineering. *IEEE Software*, 4(5):19–25, September 1987.

[127] George J. Milne. Design for verifiability. In M. Leeser and G. Brown, editors, *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 1–13. In *Lecture Notes in Computer Science* 408, Springer-Verlag, 5–7 July 1989. Also published as Research Report HDV–5–89, Department of Computer Science, University of Strathclyde, Glasgow, Scotland, UK.

[128] Robin Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[129] Larry Joe Morell. A theory of error-based testing. Technical Report TR–1395, University of Maryland, Department of Computer Science, College Park, Maryland, April 1994.

[130] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, NY, USA, 1979.

[131] Kshirasagar Naik and Behçet Sarikaya. Testing communication protocols. *IEEE Software*, 9(1):27–37, January 1992.

[132] Kshirasagar Naik and Behçet Sarikaya. Protocol conformance test case verification using timed-transitions. In Son T. Vuong and Samuel T. Chanson, editors, *Protocol Specification, Testing, and Verification, XIV*, Amsterdam, 1994. North-Holland.

[133] Gerald Neufeld and Son Vuong. An overview of ASN.1. *Computer Networks and ISDN Systems*, 23(5):393–415, February 1992.

[134] D. Parnas and Y.-B. Wang. The trace assertion method of module interface specification. Technical Report 89–261, Queen's University, Ontario, Canada, 1989.

[135] David L. Parnas, A. John van Schouwen, and Shu Po Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, June 1990.

[136] Alexandre Petrenko, Rachida Dssouli, and Hartmut Koenig. On evaluation of testability of protocol structures. In *Sixth International Workshop on Protocol Test Systems*, Amsterdam, 1993. North-Holland.

[137] T.F. Piatkowski. An engineering discipline for distributed protocol systems. In D. Rayner and R.W.S. Hale, editors, *Proc. IFIP Workshop on Protocol Testing—Towards Proof?, Volume 1: Specification and Validation*, pages 177–215, Teddington, Middlesex, U.K., 27–29 May 1981.

[138] Gee-Swee Poo and Boon-Ping Chai. Modularity versus efficiency in OSI system implementations. In *Proceedings of IEEE INFOCOM*, pages 950–959, Bal Harbour, FL, USA, 9–11 April 1991.

[139] Jon B. Postel. Transmission Control Protocol. DARPA Internet Program Protocol Specification. NIC RFC–793, September 1981.

[140] R. Probert and K. Saleh. Synthesis of communication protocols: Survey and assessment. *IEEE Transactions on Computers*, C-40(4):468–476, April 1991.

[141] R.L. Probert and O. Monkewich. TTCN: The international notation for specifying tests of communications systems. *Computer Networks and ISDN Systems*, 23(5):417–438, February 1992.

[142] Robert L. Probert and Cecilia Geldrez. Communications systems design for testability: Grey-box testing. Technical Report 90–34, Department of Computer Science, University of Ottawa, Ottawa, Canada, July 1990.

[143] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[144] John H. Reif and Scott A. Smolka. The complexity of reachability in distributed communicating processes. *Acta Informatica*, 25(3):333–354, 1988.

[145] D. Richardson and M. Thomas. The RELAY model of error detection and its application. In *Proceedings of the ACM SIGSOFT/IEEE 2nd Workshop on Software Testing, Analysis, and Verification*, Banff, Alberta, Canada, July 1988.

[146] Marc Riese and Kateel Vijayananda. Characterisation of protocol faults based on experience in interoperability testing. In *Proceedings of the 2nd International Symposium on Interworking*, pages 71–80, Sophia Antipolis, France, 4–6 May 1994. Appears in Interoperability in Broadband Networks, 1994.

[147] Brigitte Rozoy. On distributed languages and models for distributed computation. In Irene Guessarian, editor, *Semantics of systems of concurrent processes: LITP Spring School on Theoretical Computer Science*, pages 434–456, La Roche Posay, France, 23–27 April 1990. In *Lecture Notes in Computer Science* 469.

[148] Ronitt Rubinfeld. A mathematical theory of self-checking, self-testing and self-correcting programs. Technical Report 90–054, International Computer Science Institute, Berkeley, CA, USA, October 1990.

[149] John Rushby. Formal methods and the certification of critical systems. Technical Report CSL–93–7, SRI International, Computer Science Laboratory, Menlo Park, CA, USA, 1993.

[150] K. Sabnani and A. Netravali. A high speed transport protocol for datagram/virtual circuit networks. In *Proceedings of the SIGCOMM Symposium on Communications Architectures, Protocols and Applications*, pages 146–157, Austin, TX, USA, 19–22 September 1989.

[151] Kassem Saleh. Testability-directed service definitions and their synthesis. In *Proc. of the Eleventh IEEE Phoenix Conference on Computers and Communications*, Arizona, USA, March 1992.

[152] Behçet Sarikaya. Conformance testing: Architectures and test sequences. *Computer Networks and ISDN Systems*, 17:111–126, 1989.

[153] Behçet Sarikaya and Gregor von Bochmann. Synchronization and specification issues in protocol testing. *IEEE Transactions on Communications*, COM-32(4):389–395, April 1984.

[154] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[155] Beth A. Schroeder. On-line monitoring: A tutorial. *Computer*, 28(6):72–78, June 1995.

[156] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[157] Deepinder P. Sidhu. Protocol testing: The first ten years, The next ten years. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Proc. IFIP Protocol Specification, Testing and Verification, X*, pages 47–68, Amsterdam, 1990. North-Holland.

[158] Deepinder P. Sidhu and Ting-Kau Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, SE-15(4):413–426, April 1989.

[159] Arne Skou. *Validation of Concurrent Processes with Emphasis on Testing.* PhD thesis, Department of Mathematics and Computer Science, University of Aalborg, Aalborg, Denmark, December 1989.

[160] Anthony M. Sloane. Extending object interfaces for debugging. Technical Report CU-CS-644-93, University of Colorado, Department of Computer Science, Boulder, CO, USA, February 1993.

[161] M. Spezialetti and J. P. Kearns. A general approach to recognizing event occurrences in distributed computations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 300–307, San Jose, CA, USA, 13–17 June 1988.

[162] W. Timothy Strayer, Bert J. Dempsey, and Alfred C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley, Reading, MA, USA, 1992.

[163] Gregory F. Sullivan and Gerald M. Masson. Using certification trails to achieve software fault tolerance. In *20th International Symposium on Fault-Tolerant Computing Symposium*, pages 423–431, Chapel Hill, NC, USA, 26–28 Jun 1990.

[164] Gregory F. Sullivan and Gerald M. Masson. Certification trails for data structure. In *21st International Symposium on Fault-Tolerant Computing Symposium*, pages 240–247, Montreal, Quebec, Canada, 25–17 Jun 1991.

[165] Gregory F. Sullivan, Dwight S. Wilson, and Gerald M. Massson. Certification trails and software design for testability. In *Proc. of the 24th IEEE International Test Conference*, pages 200–209, Baltimore, MD, USA, 17–21 October 1993.

[166] Liba Svobodova. Implementing OSI systems. *IEEE Journal on Selected Areas in Communications*, 7(7):1115–1130, September 1989.

[167] Richard H. Thayer and Mildred C. Thayer. Glossary. In Richard H. Thayer and Merlin Dorfman, editors, *System and Software Requirements Engineering*, pages 605–676. IEEE Computer Society Press, 1990.

[168] Frank F. Tsui. *LSI/VLSI Testability Design.* McGraw-Hill, New York, NY, 1987.

[169] A. Ulrich and S.T. Chanson. An approach to testing distributed software systems. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification, XV*, Warsaw, Poland, 1995. Chapman & Hall, UK.

[170] A.P. Varvitsiotis and G.I. Stassinopoulos. Extending ASN.1 into a full-fledged constraint language in the context of OSI protocol conformance testing. *Computer Networks and ISDN Systems*, 25(11):1243–1263, June 1993.

[171] S. Venkatesan and Brahma Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, February 1995.

[172] C.A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In Sudhir Aggarwal and Krishan Sabnani, editors, *Protocol Specification, Testing, and Verification, VIII*, pages 189–204, Amsterdam, 1988. North-Holland.

[173] Chris A. Vissers. FDTs for open distributed systems: A restrospective and a prospective view. In Luigi Logrippo, Robert L. Probert, and Hasan Ural, editors, *Protocol Specification, Testing, and Verification, X*, Amsterdam, 1990. North-Holland.

[174] Chris A. Vissers and Luigi Logrippo. The importance of the service concept in the design of data communications protocols. In Michel Diaz, editor, *Protocol Specification, Testing, and Verification, V*, pages 3–19, Amsterdam, 1985. North-Holland.

[175] Jeffrey Voas. A dynamic failure model for predicting the impact that a program location has on the program. In A. van Lamsweerde and A. Fugetta, editors, *Proc. of the Third European Software Engineering Conference*, pages 308–331, Milan, Italy, 21–24 October 1991. In *Lecture Notes in Computer Science* 550.

[176] Jeffrey M. Voas. Factors that affect software testability. In *Proc. of the Ninth Pacific Northwest Software Quality Conference*, pages 235–247, Portland, OR, USA, 7–8 October 1991.

[177] Jeffrey M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, SE-18(8):717–727, August 1992.

[178] Jeffrey M. Voas and Keith W. Miller. Improving the software development process using testability research. In *Proc. of the Third International Symposium on Software Reliability Engineering*, pages 114–121, Research Triangle Park, NC, USA, 7–10 October 1992.

[179] Jeffrey M. Voas and Keith W. Miller. Semantic metrics for software testability. *The Journal of Systems and Software*, 20(3):207–216, March 1993.

[180] Jeffrey M. Voas, Jeffery E. Payne, and Keith W. Miller. Designing programs that are less likely to hide faults. *The Journal of Systems and Software*, 20(1):93–100, January 1993.

[181] Son T. Vuong and Jadranka Curgus. On test coverage metrics for communication protocols. In *Fourth International Workshop on Protocol Test Systems*, 1991.

[182] S.T. Vuong, A.A.F. Loureiro, and S.T. Chanson. A framework for the design for testability of communication protocols. In *Sixth International Workshop on Protocol Test Systems*, pages 89–108, Pau, France, 28–30 September 1993. Published as IFIP Transactions C: Communication Systems C–19 1994, North Holland.

[183] S.T. Vuong, A.A.F. Loureiro, L. Takeuchi, W. Zhu, and S. Zhang. FDDI MAC protocol specification using formal description techniques. In *Proceedings of International Symposium on Communications*, pages 731–735, Tainan, Taiwan, R.O.C., 9–13 December 1991.

[184] Brian Waldecker and Vijay K. Garg. Detection of strong predicates in distributed programs. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 692–699, Dallas, Texas, USA, 2–5 December 1991.

[185] R. W. Watson. Delta-t protocols specification. Lawrence Livermore Laboratory, California, USA, 15 April 1983.

[186] Werner Schütz. On the Testability of Distributed Real-Time Systems. In *Proc. of the Tenth Symposium on Reliable Distributed Systems*, pages 52–61, Pisa, Italy, 30 Sep–2 Oct 1991. Also published as Research Report 6, Technical University of Vienna, Vienna, Austria, 1991.

[187] C.H. West. General technique for communications protocol validation. *IBM Journal of Research and Development*, 22:393–404, April 1978.

[188] Colin H. West. Protocol validation in complex systems. In *Proceedings of ACM SIGCOMM '89*, pages 303–312, Austin, TX, USA, September 1989. Also published as Computer Communication Review, 19(4), September 1989.

[189] Lee White and Hareton K.N. Leung. Regression testability. *IEEE Micro*, 12(2):81–84, April 1992.

[190] Maurice V. Wilkes. *Memoirs of a Computer Pioneer.* MIT Press series in the history of computing. The MIT Press, Cambridge, MA, USA, 1985.

[191] Thomas W. Williams and Kenneth P. Parker. Design for testability—A survey. *IEEE Transactions on Computers*, C-31(1):2–15, January 1982.

[192] Hua-Long Yu. Testability-directed specification of communications software. Master's thesis, Department of Computer Science, University of Ottawa, Ottawa, Canada, 1991.

[193] P. Zafiropulo, C.H. West, D.D. Cowan, and D. Brand. Toward analyzing and synthesizing protocols. *IEEE Transactions on Communications*, COM-28:651–660, April 1980.

[194] Hua-Xin Zeng, Samuel T. Chanson, and Brian R. Smith. On ferry clip approaches in protocol testing. *Computer Networks and ISDN Systems*, 17(2):77–88, July 1989.

[195] H. Zimmermann. OSI reference model: The ISO model of architecture for open systems interconnections. *IEEE Transactions on Communications*, COM-28(4):651–660, April 1980.

Appendix A

Another example of self-stabilizing protocol

A.1 Transmission Control Protocol

The TCP (Transmission Control Protocol) [139] is the connection-oriented transport protocol in the Internet suite. It offers a reliable end-to-end data transfer service equivalent to the ISO Class 4 transport protocol.

A.1.1 Elements related to the protocol specification

Formal method. The CEFSM that represents the TCP is shown in Figure A.1. Both the initiator and the responder users are shown in the same figure.

Type of communication among entities. The initiator and responder users are autonomous to initiate a communication.

Timeout actions. The only state that has explicitly a timeout action is TIMEWAIT.

A.1.2 Applying the algorithm to introduce the self-stabilizing features into the protocol specification

Phases, paths, and lock-step mode. The TCP protocol as shown in Figure A.1 is comprised of the connection, data transfer, and disconnection phases and they are treated together in the CEFSM model. The connection phase starts in the state CLOSED, the data transfer phase in the state ESTABLISHED, and the disconnection phase also in the state ESTABLISHED. The set of initial states can be defined as $IS = \{$CLOSED, LISTEN, ESTABLISHED$\}$.

Figure A.2-(a) shows only the transitions in the protocol. There are 25 distinct paths in the CEFSM of the TCP and they are listed in Figure A.2-(b). The edges that represent the last transition of these paths are A, B, G, H, I, J, P, and T. The lock-step mode will be introduced in these edges.

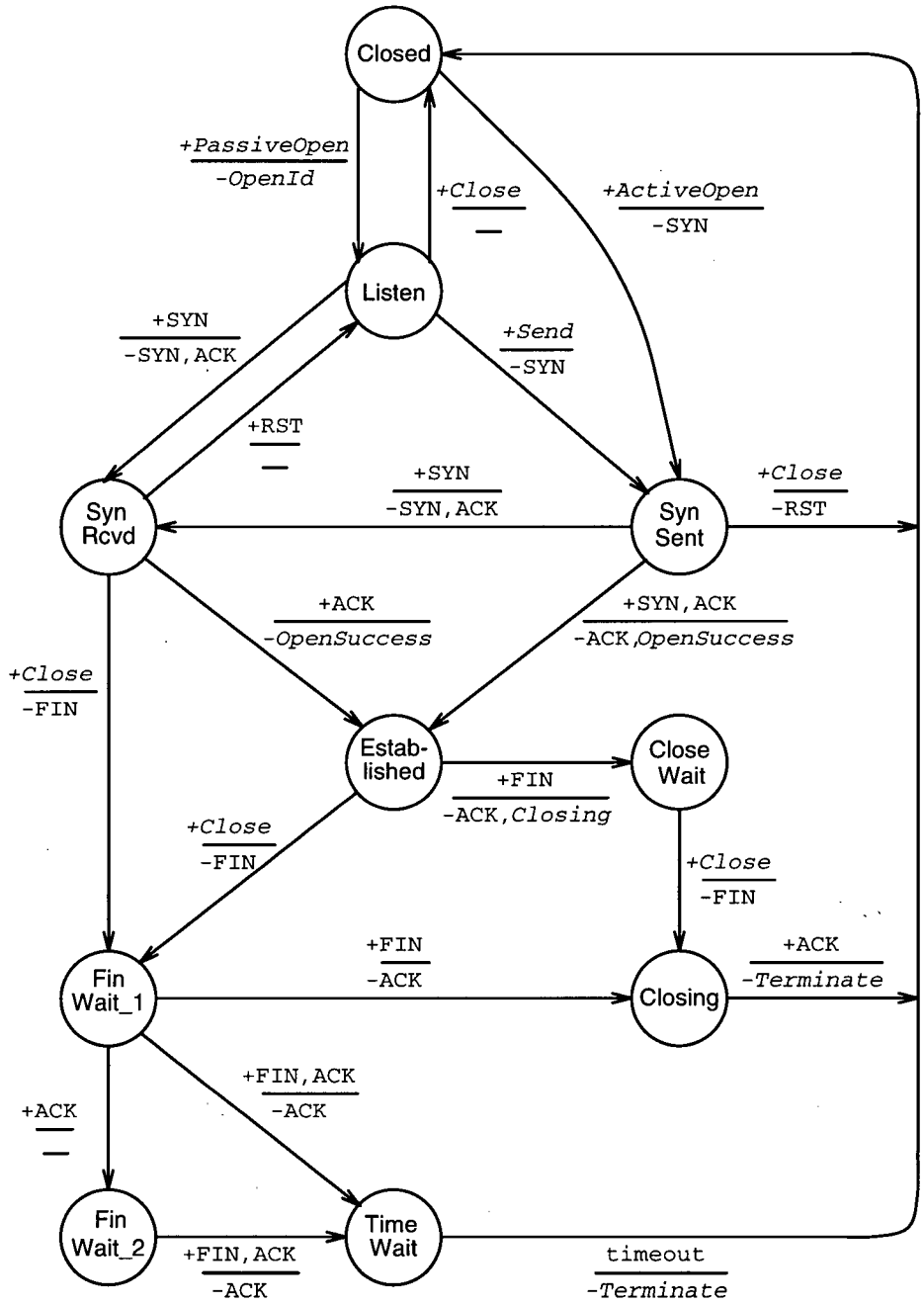
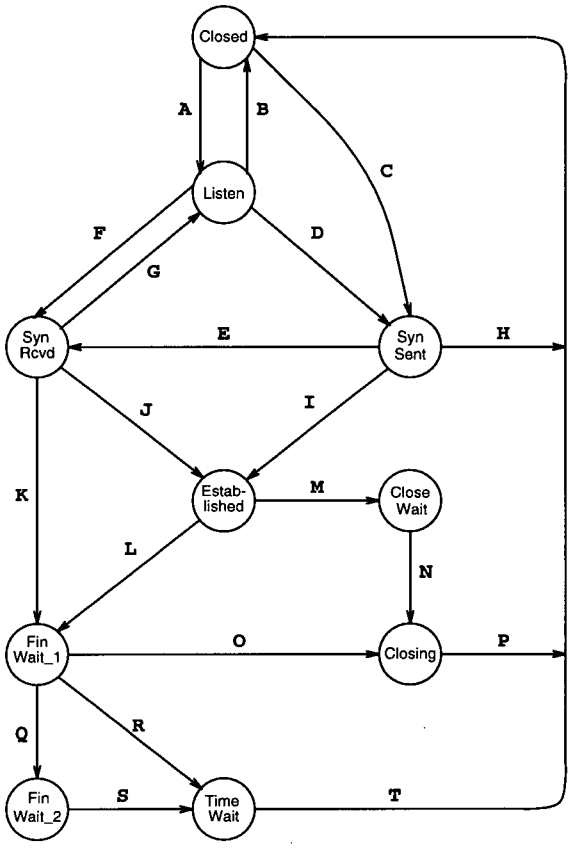


Figure A.1: Transmission Control Protocol–CEFSM model.

Timeout actions. The following notation will be used to represent the 10 states of the TCP:



(a) Transitions in the TCP.

A	LOP
B	MNP
AFG	AFKOP
CEG	CEKOP
ADEG	ADEKOP
CH	LRT
ADH	LQST
CI	AFKRT
ADI	CEKRT
AFJ	ADEKRT
CEJ	AFKQST
ADEJ	CEKQST
	ADEKQST

(b) Paths in the CEFSM of the TCP.

Figure A.2: Transitions and paths in the TCP.

Closed	Cd
Listen	L
SynRcvd	SR
SynSent	SS
Established	E
CloseWait	CW
FinWait_1	FW1
FinWait_2	FW2
Closing	Cg
TimeWait	TW

The 25 paths shown in Figure A.2 with their respective states are:

A:	Cd → L
B:	L → Cd
AFG:	Cd → L → SR → L
CEG:	Cd → SS → SR → L
ADEG:	Cd → L → SS → SR → L
CH:	Cd → SS → Cd
ADH:	Cd → L → SS → Cd
CI:	Cd → SS → E
ADI:	Cd → L → SS → E
AFJ:	Cd → L → SR → E
CEJ:	Cd → SS → SR → E
ADEJ:	Cd → L → SS → SR → E
LOP:	E → FW1 → Cg → Cd
MNP:	E → CW → Cg → Cd
AFKOP:	Cd → L → SR → FW1 → Cg → Cd
CEKOP:	Cd → SS → SR → FW1 → Cg → Cd
ADEKOP:	Cd → L → SS → SR → FW1 → Cg → Cd
LRT:	E → FW1 → TW → Cd
LQST:	E → FW1 → FW2 → TW → Cd
AFKRT:	Cd → L → SR → FW1 → TW → Cd
CEKRT:	Cd → SS → SR → FW1 → TW → Cd
ADEKRT:	Cd → L → SS → SR → FW1 → TW → Cd
AFKQST:	Cd → L → SR → FW1 → FW2 → TW → Cd
CEKQST:	Cd → SS → SR → FW1 → FW2 → TW → Cd
ADEKQST:	Cd → L → SS → SR → FW1 → FW2 → TW → Cd

The states SR, SS, CW, FW1, Cg, and FW2 must have timeout actions.