Testing Techniques

Jan Tretmans

Formal Methods & Tools Group
Faculty of Computer Science
University of Twente
The Netherlands

Copyright ©2002 by Jan Tretmans, University of Twente, The Netherlands.

Prologue

Testing is an important technique to check and control software quality. It is part of almost any software development project. Time and resources devoted to testing may vary from 30% up to 70% of a project's resources. Yet, testing is often an underexposed and underestimated part of software development. Also academic research and education in testing are rather low compared with the importance of the subject in real-life software projects. The course on *testing techniques* is intended to be a step towards bridging this gap.

Testing techniques deals with a number of topics related to software testing. The aim is that the student will get insight into some of the problems of software testing and that (s)he gets familiar with a couple of solutions to these problems. Emphasis is on techniques for testing technical software systems, such as communication software, control systems and embedded software. Established testing techniques as well as new developments will be presented. In the latter category there will be a strong focus on the use of formal methods in the software testing process. It is impossible to give a complete view of all possible testing techniques and processes in a single course. It is my hope that *testing techniques* is a reasonable compromise between giving a complete overview of the broad field of testing and dealing thoroughly with distinct testing techniques without being too superficial. There are many testing topics which are not presented in the course; for these topics I refer to standard text books on software testing.

These lecture notes are intended to support the course on *testing techniques*; they are not completely self-contained. Some topics will be more elaborately presented during the oral lectures than they are discussed in these notes, while other parts of these lecture notes serve as background information and will only be briefly mentioned during the lectures.

The lecture notes consist of two parts. The first part deals with new developments in testing, in particular with software testing based on formal methods. It is a compilation of a number of articles: [BT01, TB99, Tre96b, Tre99]. The formal methods that are used are introduced in the text, but it certainly is not an introductory text on formal methods. Some basic knowledge about formal methods is desirable. The second part deals with established testing techniques. It consists of a collection of articles which address various techniques.

Acknowledgements Numerous people contributed in some way or another to the text of Part I, either to the scientific developments on which it is based, or by means of stimulating discussions or commenting on earlier versions, for which I am grateful.

These include our partners in the *Côte de Resyste* research project, the participants in the ISO/ITU-T standardization group on "Formal Methods in Conformance Testing", testing engineers at CMG N.V., and the members of the Formal Methods and Tools group at the University of Twente. Section 1.4 is co-authored with Ed Brinksma, and Section 6.2 was written by Axel Belinfante as part of [TB99]. Of course, any remaining errors or deficiencies are solely my responsibility.

Financial support for the research reflected in Part I was provided by the Dutch Technology Foundation STW in the context of *Côte de Resyste*. The project *Côte de Resyste* (COnformance TEsting OF REactive SYSTEms) aims at improving the testing process by using formal methods. Theories, methods and tools are developed to enable fully automatic, functional testing of reactive software systems based on formal specifications. *Côte de Resyste* is a cooperation between Philips Research Laboratories Eindhoven, Lucent Technologies R&D Centre Enschede, Eindhoven University of Technology and the University of Twente; for further information see http://fmt.cs.utwente.nl/cdr.

These lecture notes on *testing techniques* are still under development. I would like to invite readers to test this text thoroughly and I will be grateful for any detected defects, comments, unclearities and suggestions.

Hengelo Ov., February 2002

Jan Tretmans

tretmans@cs.utwente.nl

Contents

Part I

Testing with Formal Methods

Chapter 1

Introduction

1.1 General

Software testing Software quality is an issue that currently attracts a lot of attention. Software invades everywhere in our society and life and we are increasingly dependent on it. Moreover, the complexity of software is still growing. Consequently, the quality, functional correctness and reliability of software is an issue of increasing importance and growing concern. Systematic testing of software plays an important rôle in the quest for improved quality.

Software testing involves experimentally and systematically checking the correctness of software. It is performed by applying test experiments to a software system, by making observations during the execution of the tests, and by subsequently assigning a verdict about the correct functioning of the system. The correctness criterion against which the system is to be tested should be given in some kind of system specification. The specification prescribes what the system has to do and what not, and, consequently, constitutes the basis for any testing activity.

Despite its importance, testing is often an under-exposed phase in the software development process. Testing turns out to be expensive, difficult and problematic while, on the other hand, research and development in testing have been rather immature. Testing methodology is mostly governed by heuristics. Fortunately, this situation is gradually improving. Triggered by the quest for improved quality and imposed by increased product liability, testing is considered more important and treated more seriously. Being a software tester is becoming a true profession.

One line of developments for improving the testing process is the use of *formal methods*. This text deals with software testing based on formal methods.

Formal methods Currently, most system specifications are written in natural languages, such as English, Dutch or Greek. Although such informal specifications are easily accessible, they are often incomplete and liable to different and possibly inconsistent interpretations. Such ambiguities are not a good basis for testing: if it is not clear what a system shall do, it is difficult to test whether it does what it should do.

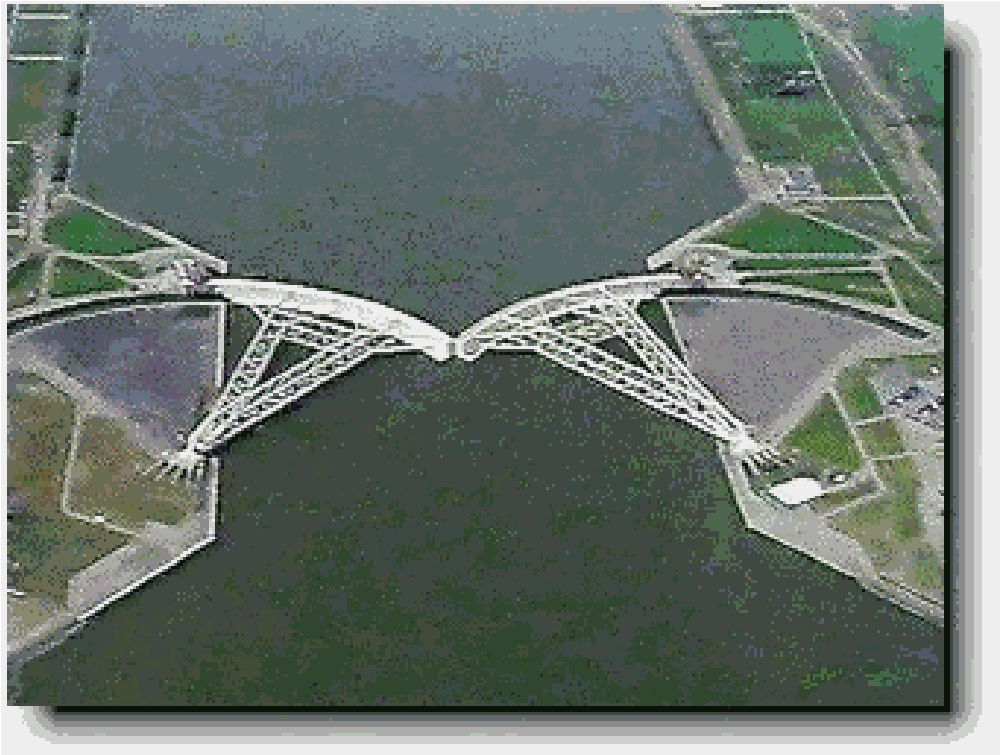


Figure 1.1: The *Maeslant Kering* is a movable storm surge barrier which is completely controlled by software. At the top of the figure the Nieuwe Waterweg, which is approximately 300 m. wide, flows to the North Sea; in the bottom direction is Rotterdam. To protect Rotterdam from flooding the software which controls the opening and closing of the barrier must be very reliable. In the design of the software the formal methods *Z* and PROMELA were used [TWC01]. Testing was performed based on the formal models [GWT98].

With formal methods systems are specified and modelled by applying techniques from mathematics and logic. Such formal specifications and models have a precise, unambiguous semantics, which enables the analysis of systems and the reasoning about them with mathematical precision and rigour. Moreover, formal languages are more easily amenable to automatic processing by means of tools. For example, tools exist that are able to verify fully automatically the absence of deadlock based on a formal description of the design.

Until recently formal methods were a merely academic topic, but now their use in industrial software development is increasing, in particular for safety critical systems and for telecommunication software. One particular, safety critical, Dutch project where formal methods have been used successfully is the control system for the *Maeslant Kering*, see Figure 1.1.

Testing with formal methods A formal specification is a precise, complete, consistent and unambiguous basis for design and code development as well as for testing. This is a first big advantage in contrast with traditional testing processes where such a

basis for testing is often lacking. A second advantage of the use of formal specifications for testing is their suitability to automatic processing by means of tools. Algorithms have been developed which derive tests from a formal specification. These algorithms have a sound theoretical underpinning. Moreover, they have been implemented in tools leading to automatic, faster and less error-prone test generation. This opens the way towards completely automatic testing where the system under test and its formal specification are the only required prerequisites. Formal methods provide a rigorous and sound basis for algorithmic and automatic generation of tests. Tests can be formally proved to be valid, i.e., they test what should be tested, and only that.

Scope This text deals with testing based on formal methods. In particular, it deals with theory, methods and tools for *functional testing* of *reactive software systems* based on *transition system specifications*.

Functional testing involves checking the correct behaviour of a system: does the system do what it should do – as opposed to, e.g., testing the performance or robustness of a system.

Reactive systems are mostly technical, event-driven systems in which stimulus/response behaviour is important. Concurrency, distribution and nondeterministic behaviour usually play an important rôle in such systems. Examples are embedded systems, communication protocols and process control software. Administrative systems are typically not reactive systems.

A transition system specification is a prescription of required behaviour in the form of a formal expression for which the semantics can be expressed as a *labelled transition system*. Typical examples of such formal languages are *process algebras*, but also many other languages can be expressed in terms of labelled transition systems.

Overview The remainder of Chapter 1 gives introductions to software testing and testing based on formal methods, it presents some bibliographic information and it mentions some other approaches which are not further discussed in this text. Chapter 2 presents a framework for testing with formal methods. This framework is instantiated and elaborated for labelled transition systems in Chapters 4 and 5, but before that labelled transition systems themselves are discussed in Chapter 3. Chapter 6 presents some tools based on this theory and some applications of these tools.

1.2 Software Testing

Testing Testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Tests are applied to the implementation under test in a controlled environment, and, based on observations made during the execution of the tests, a verdict about the correct functioning of the implementation is given. The correctness criterion that is to be tested is given by the system specification. The specification prescribes what the system shall do and what not, and, consequently, constitutes the basis for any testing activity. Testing is an important technique to increase confidence in the quality of a computing system. In almost any software development trajectory some form of testing is included.

Sorts of testing There are many different kinds of testing. In the first place, different aspects of system behaviour can be tested: Does the system have the intended functionality and does it comply with its functional specification (functional tests or conformance tests)? Does the system work as fast as required (performance tests)? How does the system react if its environment shows unexpected or strange behaviour (robustness tests)? Can the system cope with heavy loads (stress testing)? How long can we rely on the correct functioning of the system (reliability tests)? What is the availability of the system (availability tests)? Can an external intruder easily read or modify data (security testing)?

Moreover, testing can be applied at different levels of abstraction and for different levels of (sub-)systems: individual functions, modules, combinations of modules, subsystems and complete systems can all be tested.

Another distinction can be made according to the parties or persons performing (or responsible for) testing. In this dimension there are, for example, system developer tests, factory acceptance tests, user acceptance tests, operational acceptance tests, and third party (independent) tests, e.g., for certification.

A very common distinction is the one between black box and white box testing. In black box testing, or functional testing, only the outside of the system under test is known to the tester. In white box testing, also the internal structure of the system is known and this knowledge can be used by the tester. Naturally, the distinction between black and white box testing leads to many gradations of grey box testing, e.g., when the module structure of a system is known, but not the code of each module.

From now on, we concentrate on black box, functional testing. We do not care about the level of (sub-)systems or who is performing the testing. Key points are that there is a system implementation exhibiting behaviour and that there is a specification. The specification is a prescription of what the system should do; the goal of testing is to check, by means of testing, whether the implemented system indeed satisfies this prescription. We will also refer to this kind of testing as *conformance testing*.

Confusion of tongues We use a rather narrow description of the term *testing*. In other environments the term is sometimes also used to refer to performing static checks on the program code, e.g., checking declarations of variables using a static checker, or code inspections. This kind of testing is then denoted by *static testing*. However, we restrict to *dynamic testing*, i.e., testing consisting of really executing the implemented system, as described above. Another broader use of the term testing is to include *monitoring*. Monitoring is also referred to as *passive testing* as opposed to *active testing* as described above, where the tester has active control over the test environment, and a set of predefined tests is executed. A third extension of the term testing, sometimes made, is to include all checking activities in the whole software development trajectory, e.g., reviews and inspections.

Aim of testing Testing consists of trying to expose potential errors. Different kinds of errors can be exposed. In particular, errors emanating from different phases of the development trajectory may be discovered: errors made during the requirements capturing phase (e.g., during acceptance testing), during the specification phase, during design, or implementation. Of course, errors should be discovered as soon as possible during the development trajectory, since it is a well-known fact that the later errors

are exposed the more expensive their repair will be. In the ideal case, errors made in a particular development phase are detected during the verification and validation activity of that phase. In this ideal situation, only coding errors would be found during system testing (since the design would be completely validated and correct). Unfortunately, this is usually not the case: also requirement-, specification-, and design errors are normally exposed during system testing. In many cases these errors are not really errors, but can be traced back to impreciseness, ambiguities, or incompleteness in the requirement-, specification, or design documents.

The testing process In the testing process different phases can be distinguished. A crude view distinguishes between *test generation* and *test execution*. Test generation involves analysis of the specification and determination of which functionalities will be tested, determining how these can be tested, and developing and specifying test scripts. Test execution involves the development of a test environment in which the test scripts can be executed, the actual execution of the test scripts and analysis of the execution results and the assignment of a verdict about the well-functioning of the implementation under test.

In a more detailed view on the testing process more activities can be distinguished: test organization, test analysis, test specification, test implementation, test application, test result analysis and test maintenance.

Test organization The whole testing process should be well organized, well structured and well planned. Of course, this applies to any activity. Test organization is the overall activity that observes and controls all other testing activities. It involves developing a test strategy, the break-down into different activities and tasks, planning of all activities and tasks, allocation of time and resources, control, reporting and checking of each activity and task, and keeping track of systems and delivered products. Especially important for testing is version management: which test sets are available, which versions, to which system versions do they apply, which test tools do they require, and on which versions of the requirements, specification or design documents are they based. Test organization, although very important, is not the main focus point of this text.

Test analysis Test analysis consists of analysing what should be tested: the system under test and the requirements and specifications against which it should be tested. Moreover, the sorts of testing, the aspects of testing, who will do the testing and the level of abstraction are determined.

It is important to start test analysis already during the requirements capturing and specification phases of system development. In this way test analysis allows to check whether the requirements and specifications are consistent, complete, non-ambiguous, and, most important for testing, testable.

Test specification During test specification a set of test cases (a test suite) is developed and precisely specified: sequences of test events are composed, inputs are devised and the corresponding expected outputs are documented. The test suite should test the system thoroughly, completely and in every aspect, referring to the relevant requirement-, specification-, or design documents.

For a test suite to make sense it is important that it is based on a well-defined and identified requirement-, specification- or design document. A test suite in isolation does not have a meaning: it only specifies how to test, not what is being tested. This latter remark applies especially to system modifications: re-testing a system after a modification only makes sense if the appropriate requirement-, specification- or design document has been updated accordingly to express precisely what has to be re-tested.

In a well-designed test suite there is a balance between the thoroughness of the test suite and the cost and time it takes to develop and execute the test suite. Such a balance should be based on analysis of the risks of remaining, not discovered errors in the system, and on the marginal cost of additional testing.

The notation for specifying test cases should satisfy certain requirements. Firstly, it is important that the tests are described precisely and completely, so that no ambiguity arises when they are implemented and executed. Secondly, each test should refer clearly to the requirement(s) it tests. Thirdly, test cases should be represented in an accessible format so that also non-experts can analyse the tests and what they test. Fourthly, test cases should preferably be represented in such a way that subsequent test implementation and execution can be performed in an automatic way as much as possible. Fifthly, the test notation should be abstract enough to allow for test specification to start as soon as the relevant requirement-, system specification- or design documents are available, without the need to wait for the system implementation to be finished.

Test implementation Test implementation concerns the preparation of the environment in which the tests will be run. This involves the choice or development of test systems, test probes, devices to stimulate or observe the system under test, implementation of the generated test specifications, etc., in such a way that the specified test suites can be efficiently and effectively run and test results can be observed and logged for later analysis.

Test application Test application consists of systematically running all the test cases in a test suite to the system under test and logging all important events that occur during the test runs. Since a test suite can be large, efficiently running them is of major importance. Moreover, careful logging of test events in a format that is accessible for later analysis, is important.

Test runs should be repeatable: later execution of the same test case on the same system under test should deliver analogous test results (modulo nondeterminism in the system under test).

Test result analysis When all test events have been carefully registered they can be analysed for compliance with the expected results, so that a verdict about the system's well-functioning can be assigned.

Test maintenance Maintenance of test suites is a very important aspect of the testing process, but it is often underestimated. Test maintenance involves storing and documenting the products of the testing process (the *testware*) for later reuse: the test scripts, test environments, used test tools, relations between test sets and versions of

specifications and implementations, etc. The aim is to make the testing process repeatable and reusable, in particular for regression testing. Regression testing is the re-testing of unmodified functionality in case of a modification of the system. It is one of the most expensive (and thus often deliberately "forgotten") aspects of testing.

Test automation Testing is an expensive, time-consuming and labour-intensive process. Moreover, testing is (should be) repeated each time a system is modified. Hence, testing would be an ideal candidate for automation. Consequently, there are many test tools available nowadays. Most of these test tools support the test execution process. This includes the execution, systematic storage and re-execution of specified test cases. Test cases have to be written down – manually – in a special language for test scripts which is usually tool specific. These test scripts can then be executed automatically. An alternative approach is *capture & replay*. Capture & replay tools (record & playback tools) are able to record user actions at a (graphical) user interface, such as keyboard and mouse actions, in order to replay these actions at a later point in time. In this way a recorded test can be replayed several times. The advantages of automation of test execution are mainly achieved when tests have to be re-executed several times, e.g., during regression testing.

Test execution tools do not help in developing the test cases. Test cases are usually developed by clever humans, who, while reading and studying specifications, think hard about what to test and about how to write test scripts that test what they want to test. There are not many tools available that can help with, let alone automate, the generation of good tests from specifications. Tools that do exist are, e.g., tools which are able to generate large amounts of input test data. However, these tools are mainly used for performance and stress tests and hence are not further discussed. Also some tools exist that are able to generate a set of tests with the same structure based on a template of a test case by only varying the input parameters in this template. Generation of tests from formal specifications will be the topic of the remaining chapters of this text.

For other activities of the testing process other tools, not specific to testing, can be used, e.g., to relate test cases to the requirements that they test, standard requirements management tools can be used. The main functionality of such tools is to relate high level system requirements to (lower level) sub-system requirements and to relate each requirement to (a) test case(s) that tests that requirement.

A kind of test tools which are used during test execution, but which (should) influence test generation, are code coverage tools. Code coverage tools calculate the percentage of the system code executed during test execution according to some criterion, e.g., "all paths", "all statements", or "all definition-usage combinations" of variables. They give an indication about the completeness of a set of tests. Note that this notion of completeness refers to the implemented code (white box testing); it does not say anything about the extent to which the requirements or the specification were covered.

Some testing problems During testing many problems may be encountered. There can be organizational problems, e.g., what to do when during the testing process, technical problems, e.g., when there are insufficient techniques and support for test specification or test execution leading to a laborious, manual and error-prone testing process, financial problems, e.g., when testing takes too much effort – in some software

developments projects testing may consume up to 50% of the project resources – or planning problems, e.g., when the testing phase gets jammed between moving code delivery dates and fixed final custom delivery dates. Dealing with all these problems is not always simple.

An important class of problems attributed to testing turn out to be actually problems of system specification. Many problems in the testing process, in particular during test generation, occur because specifications are unclear, imprecise, incomplete and ambiguous. Without a specification which clearly, precisely, completely and unambiguously prescribes how a system implementation shall behave, any testing will be very difficult because it is unclear what to test. A bad system specification as starting point for the testing process usually leads to problems such as difficulties of interpretation and required clarifications of the specifyer's intentions. This leads to reworking of the specification during the testing phase of software development.

Automation of testing activities may help in alleviating some of the other problems, but certainly test automation will never solve all problems. Moreover, test automation only helps if the testing process has already reached a certain level of maturity: automating chaos only gives more chaos, or, stated differently: a fool with a tool is still a fool. Successful automation may help in making the testing process faster, in making it less susceptible to human error by automating routine or error-prone tasks, and in making it more reproducible by making it less dependent on human interpretation.

Moreover, the laborious and manual test generation phase is difficult to automate. In the first place, many current-day specifications are unclear, incomplete, imprecise and ambiguous, as explained above, which is not a good starting point for systematic development of test cases. In the second place, current-day specifications are written in natural language, e.g., English, German, etc. Natural language specifications are not easily amenable to tools for automatic derivation of the test cases; formal language specifications are.

1.3 Testing with Formal Methods

Formal conformance testing Conformance testing with formal methods refers to checking functional correctness, by means of testing, of a black-box system under test with respect to a formal system specification, i.e., a specification given in a language with a formal semantics. The formality of the specification implies that the correct behaviour of the system under test is formally, i.e., precisely, defined.

As stated, this text concentrates on formal specification languages that are intended to specify functional behaviour of reactive systems, in particular, specification languages with semantics expressed in terms of labelled transition systems, e.g., LOTOS [ISO89], PROMELA [Hol91], SDL [CCI92], process algebras [Mil89], and many others for which, with more or less effort, a mapping to labelled transition systems can be made.

The formal conformance testing process Also in a process of conformance testing with formal methods the main phases are test generation and test execution. The difference is that now a formal specification is the starting point for the generation of test cases. This allows to automate the generation phase: test cases can be derived algorithmically from a formal specification following a well-defined and precisely spec-

ified algorithm. For well-definedness of the algorithm it is necessary that it is precisely defined what (formal) correctness is. Well-defined test generation algorithms guarantee that tests are *valid*, i.e., that tests really test what they should test, and only that.

Also without automatic test generation the use of formal specifications can be very beneficial for testing. The improved clarity, preciseness, consistency and completeness of a formal specification make that test cases can be generated efficiently, effectively and systematically from a formal specification. Even if all test generation is manual and not supported by tools, an improvement in quality and costs of testing can be achieved, as has been reported in [GWT98] for testing the control software of the *Maeslant Kering*, see Figure 1.1.

Unlike test generation, test execution is not so much influenced by the use of formal methods. Existing test execution tools can be used, no matter how the test scripts were generated. On the other hand, analysis of results and verdict assignment are simplified when using a formal specification: comparison of expected and observed responses is more easily performed.

Moreover, automatic generation of tests allows combining test generation and test execution: tests can be executed while they are generated. We call this way of interleaved test generation and test execution *on-the-fly* testing. It is the way that the test tool TorX works; it will be discussed in Chapter 6.

Formal methods versus testing During the last decades much theoretical research in computing science has been devoted to formal methods. This research has resulted in many formal languages and in verification techniques, supported by prototype tools, to verify properties of high-level, formal system descriptions. Although these methods are based on sound mathematical theories, there are not many systems developed nowadays for which correctness is completely formally verified using these methods.

On the other hand, the current practice of checking correctness of computing systems is still based on the more informal and pragmatic technique of testing. An implementation is subjected to a number of tests which have been obtained in an ad-hoc or heuristic manner. A formal, underlying theory for testing is mostly lacking.

The combination of testing and formal methods is not very often made. Sometimes it is claimed that formally verifying computer programs would make testing superfluous, and that, from a formal point of view, testing is inferior as a way of assessing correctness. Also, some people cannot imagine how the practical, operational, and 'dirty-hands' approach of testing could be combined with the mathematical and 'clean' way of verification using formal methods. Moreover, the classical biases against the use of formal verification methods, such as that formal methods are not practical, that they are not applicable to any real system but very simple toy systems, and that they require a profound mathematical training, do not help in making test engineers adopt formal methods.

Fortunately, views are changing. Academic research on testing is increasing, and even the most formal verifyer admits that a formally verified system should still be tested. (Because: Who verified the compiler? And the operating system? And who verified the verifyer?). On the other hand, formal methods are used in more and more software projects, in particular for safety critical systems, and also the view that a formal specification can be beneficial during testing is getting more support.

Formal testing and verification Formal testing and formal verification are complementary techniques for analysis and checking of correctness of systems. While formal verification aims at proving properties about systems by formal manipulation on a mathematical model of the system, testing is performed by exercising the real, executing implementation (or an executable simulation model). Verification can give certainty about satisfaction of a required property, but this certainty only applies to the model of the system: any verification is only as good as the validity of the system model. Testing, being based on observing only a small subset of all possible instances of system behaviour, can never be complete: testing can only show the presence of errors, not their absence. But since testing can be applied to the real implementation, it is useful in those cases when a valid and reliable model of the system is difficult to build due to complexity, when the complete system is a combination of formal parts and parts which cannot be formally modelled (e.g., physical devices), when the model is proprietary (e.g., third party testing), when the validity of a constructed model is to be checked with respect to the physical implementation, or when a physical implementation is checked for conformance to its model.

One of the aims of this text is to strengthen the view that testing and verification, whether formal or not, are complementary techniques. To that purpose, this text discusses how testing can be performed based on formal specifications, and how advantage can be obtained in terms of precision, clarity and consistency of the testing process by adopting this formal approach. Also, it will be shown how the use of formal methods helps automating the testing process, in particular the automated generation of tests from formal specifications.

1.4 Bibliographic Notes

Introduction Labelled transition system based test theory has made remarkable progress over the past 15 years. From a theoretically interesting approach to the semantics of reactive systems it has developed into a field where testing theory is (slowly) narrowing the gap with testing practice. In particular, new test generation algorithms are being designed that can be used in realistic situations whilst maintaining a sound theoretical basis. This section presents an annotated bibliography of labelled transition system based test theory in its historical perspective.

Formal testing theory Formal testing theory was introduced by Rocco De Nicola and Matthew Hennessy in their seminal paper [DNH84], and it was further elaborated in [DN87, Hen88]. Their original motivation was to characterize interesting formalizations of the notion of *observable behaviour* for transition systems using an idealized but intuitive formalization of testing. It contributed to the semantic theory of reactive systems, and was not intended as a theory about actual testing. One of the main behavioural preorders or *implementation relations* of [DNH84], so called *must*-testing, was in fact an alternative characterization of the standard semantic model for CSP [Hoa85], the *failures* model (at least, in the absence of infinite internal computations). Their approach actually required the formal observability of *deadlock* behaviour. This assumption was further exploited by Phillips in [Phi87], and independently by Langerak [Lan90], to define testing scenarios that allow for the testing of alternative behaviour after the observation of deadlocks, the so called *refusals* model. Abramsky showed in [Abr87] that by the introduction of still stronger assumptions (e.g., the unbounded

copying of behaviours under test), even classical behavioural equivalences like *bisimulation* [Mil89] can be characterized by testing scenarios. An overview of the theory of behavioural equivalences and preorders for transition systems, including the testing-based notions and their further extensions, can be found in the classical surveys by Van Glabbeek [Gla90, Gla93].

Test frameworks To arrive at a useful formal model for actual testing a well-defined framework of basic terminology, concepts and methods is needed. The ISO International Standard on conformance testing [ISO91] has been a very influential, informal source in this respect. A short overview can be found in [Ray87], a more critical a posteriori assessment in [Ray97]. A first attempt to come to a more formal interpretation can be found in [BAL+90], with subsequent elaborations in [TKB92, Tre94, Tre99]. This has led itself to standardization activity culminating in [ISO96], with related expositions in [CFP94, HST96].

Formal test generation The first attempts to use De Nicola-Hennessy testing theory for finding algorithms to derive tests automatically from formal specifications were made by Brinksma in [Bri87, Bri88]. This work, which used the specification language LOTOS [ISO89, BB87] as a defining notation for the transition systems, led to the so called *canonical tester* theory. This approach has led to a whole series of modifications and extensions, the main ones being [PF90, Tre90, Doo91, Dri92, Led92]. Publications on specific test generation algorithms and (in some cases) their implementations can be found in [Eer87, Ald90, Wez90, DAV93, TPB96]. All these approaches assume that the testing process can communicate *synchronously* with the system under test, i.e., that each communication can be viewed as a joint action of system and tester, as in most process algebras [Mil89, Hoa85, BK85, ISO89].

Asynchronous test contexts In practice the assumption of synchronous communication is seldom fulfilled. Initially, the problem was handled by applying existing theory on a transformation of the original formal specification. In this transformation the original transition system is 'plugged' into a context of input an output queues that enabled the asynchronous communication of inputs (test stimuli) and outputs (test responses) between tester and system under test [TV92, VTKB93]. In this way the existing theory could simply be 'lifted' to all sorts of asynchronous scenarios. The disadvantage of this approach, however, was that one had to have different transformations for each (static) communication interface. Moreover, as the communication queues were in most cases unbounded, they caused infinite state spaces even in the case of finite state specifications, which complicated the test generation problem considerably.

A major step forward was made by interpreting transition systems as descriptions of input/output-automata (I/O-automata) [LT89, Seg93]. In this model the property that test inputs cannot be blocked by the system under test is covered by the assumption of *input enabledness* (the I/O-condition), which says that in each state transitions for all input actions are defined. The impossibility to refuse test outputs is obtained by having the analogous requirement for the tester process. This semantic requirement on the system under test and on the tester allows for a uniform and simplified treatment of whole classes of asynchronous testing scenarios and has resulted in refusal and failure models that are in a certain sense simpler than their synchronous counterparts. These

ideas were first explored by Phalippou and Tretmans [Pha94a, Pha94b, Tre96a, Tre96b]. Heerink has subsequently refined these ideas by refining the I/O-condition in the sense that either all or none of the input actions are enabled, allowing for the treatment of *bounded* communication media [HT97, Hee98]. This work also allows for the treatment of multiple *Points of Control and Observation* (PCO) that exist in practical situations. The multiple I/O-paradigm for testing turns out to be a natural saturation point of the whole theory, in the sense that all relevant other implementation relations can be obtained as special cases. An overview of the main ideas can be found in [BHT97].

Test generation tools The ideas behind many of the reported test generation algorithms have been tried out in small, academic prototypes, but there are only a few examples of larger test tool developments linked to academic research in general, and transition system based testing in particular. An important point in this respect was the development of the test tool TVEDA [CGPT96]. Although not developed on the basis of a formal theory of testing, most of its underlying principles can be justified in terms of the I/O-theories of [Pha94b, Tre96b]. A more recent test tool that uses algorithms from the domain of model checking is TGV [FJJV96, FJJV97, JM99]. The ideas underlying TVEDA and TGV have been combined into TestComposer which has been incorporated into the commercial SDL tool set *Object*Geode [KJG99].

In the Dutch *Côte de Resyste* project [STW96] the tool TorX is developed which is the first larger test tool that is completely based on a formal model of conformance testing [BFV+99]. TorX accepts specifications in LOTOS and Promela; Promela is the input language for the model checker SPIN [Hol91, Spi]. The implementation of TorX uses the state-space exploration algorithms of SPIN for its Promela part [VT00], while for its LOTOS part it relies on the Cæsar-Aldebaran tool set [Gar98].

Among the other tools for formal transition system based test generation are VVT-RT which uses CSP as the input specification language [PS97], and SaMsTaG and Autolink which derive tests from SDL specifications but which have a slightly different and less formal approach [GSDD97, SEK+98]. The I/O testing paradigm is also used in hardware validation in the tool TestGen [HT99].

Current developments Current theory and tools can generate tests from transition systems based specifications, however, it is difficult to steer the test generation process and to know how much of the specification has been tested. Some of the tools use user-specified *test purposes* to steer the test generation (TGV, Autolink), while others use a random approach (TorX). The problem of finding criteria for how to select the tests with the largest chance of detecting errors is one of the major research challenges [Bri93, Pha94b, HT96, CV97, CG97, FGST02]. Among the other research items of current interest are the symbolic representation and treatment of data aspects in test generation [CJRZ01], the combination of conformance testing with real-time and performance testing [NS01], formal verification of test suites [JJM00], and distributed testing [JJKV98]. On the practical side we see increasing activities in applying the techniques and tools to realistic industrial case studies [KVZ98, BFHV01, Chr01, VBF02], while also comparison of different algorithms and tools is investigated in an experimental setting [BFV+99, DBRV+00, HFT00].

1.5 Other Approaches

Other quality improvement techniques Testing is an important technique for improving the quality of software. However, it is by no means the only technique. It is just one of the techniques, which should be used in combination with other techniques and processes to improve software quality. These other techniques include process-oriented techniques, such as a well-defined and well-managed software development process, e.g., inspired by the ideas of the *Capability Maturity Model* (CMM) [PWC95], structured design methods, version and configuration management techniques, (formal) validation and verification techniques, review and inspection techniques, etc. After all, testing only allows to prove the disfunctioning of software but it does not provide a method to make a better product. Only a carefully chosen combination of methods and techniques can assure that the final product is of sufficiently high quality.

Other testing techniques This text deals with testing with formal methods. Actually, most of current practice is non-formal. It is based on experience, heuristics and clever tester knowledge. There is an ever expanding number of text books that deal with this way of testing such as [Mye79, Bei90] and many others. The methods and techniques presented in these text books are not discussed here.

Other testing aspects This text mainly concentrates on the test generation part. There are other aspects of the testing process which are at least equally important, e.g., test execution and test organization. An important approach to test organization and management is TMap (Test Management Approach) [PV98]. Test execution is the topic of many test execution tools and books on automated testing, see e.g., [FG99, Sit00]. Analogous to *Software Process Improvement* – e.g., CMM, mentioned above – there are also approaches to *Test Process Improvement* [KP99].

Other formal approaches to testing The formal testing approach discussed in this text is based on the formalism of labelled transition systems. There are some other important approaches to formal methods based testing. The one with the longest tradition is based on Mealy-machines (also known as the FSM-approach). This approach is discussed in Part II, Chapter 3; for another overview paper, see [LY96]. The link between this theory and transition system-based testing is studied in [Tan97].

Both the FSM and the transition system-based approaches mainly deal with the dynamic aspects of system behaviour. An existing formal approach to testing the static aspects of systems, such as data structures and their operations, uses abstract data type (ADT) theory as its basis, see e.g., [Ber91, Gau95, LGA96] and [Mar95] for a corresponding tool. It is generally assumed that this approach can be combined with either of the control-oriented approaches [GJ98]. Another approach is testing based on the formal language *Z* [Hie97].

Chapter 2

A Formal Approach to Software Testing

2.1 Introduction

In Section 1.2 the software testing process was described from a traditional perspective. Conformance testing was introduced as a kind of testing where the functional behaviour of a system is systematically tested with respect to the system's specification.

In this section a framework is presented for the use of formal methods in conformance testing [BAL+90, Tre94, ISO96, Tre99]. The framework is intended to reason and argue about testing in the context of formal methods independent from any specific formal method. It can be used when an implementation is tested with respect to a formal specification of its functional behaviour.

The process of formal conformance testing is schematically depicted in Figure 2.1. Starting from a formal specification s, an implementation i is developed by a programmer; it is, for example, a C program. By means of testing we would like to check whether i conforms to s. To this extent, a test suite T_s is derived from the same specification s following a test derivation algorithm T. Subsequent execution of the test suite T_s with the implementation i leads to a verdict, either **pass** or **fail**; **pass** indicates that no evidence of non-conformance was found; **fail** indicates that an error was found.

The framework discussed in this section introduces, at a high level of abstraction, the concepts used in a formal conformance testing process and it defines a structure which allows to reason about testing in a formal way. The most important part of this is to link the informal world of implementations, tests and experiments with the formal world of specifications and models. To this extent the framework introduces the concepts of conformance, i.e., functional correctness, testing, sound and exhaustive test suites, and test derivation. All these concepts are introduced at a generic level, i.e., independent of any particular formal method. Chapters 3, 4 and 5 will show how to instantiate and apply these concepts for the particular formalism of labelled transition systems.

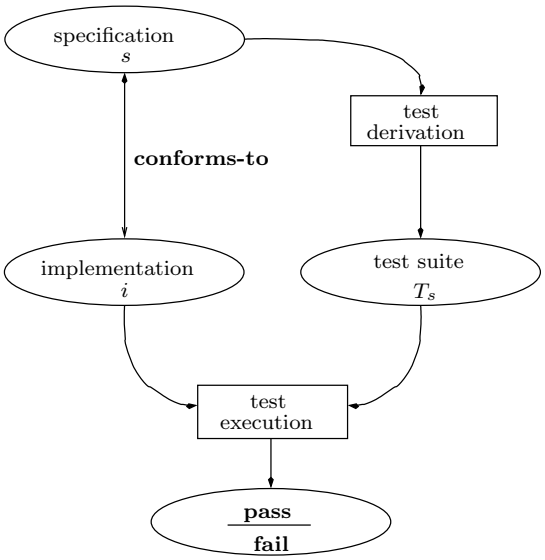


Figure 2.1: The formal conformance testing process

2.2 A Formal Framework

Conformance For talking about conformance we need implementations and specifications, see Figure 2.2. The specifications are formal, so a universe of formal specifications denoted *SPECS* is assumed. Implementations are the systems that we are going to test, henceforth they will be called IUT, which stands for implementation under test, and the class of all IUT's is denoted by *IMPS*, see Figure 2.2. So, conformance could be introduced by having a (mathematical) relation **conforms-to** $\subseteq IMPS \times SPECS$ with IUT **conforms-to** s expressing that IUT is a correct implementation of specification s.

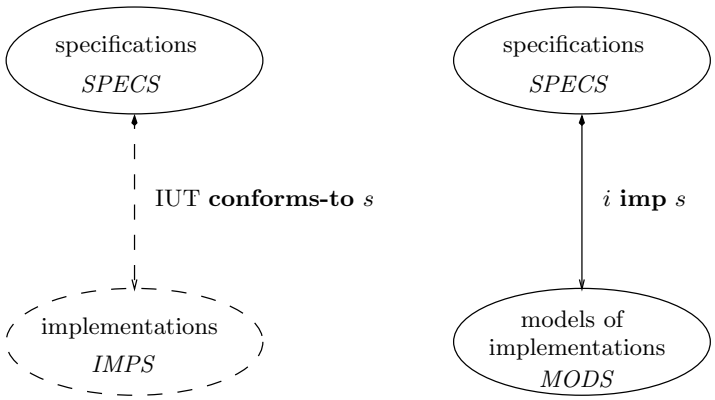


Figure 2.2: Specifications, implementations, and models of implementations

However, unlike specifications, implementations under test are real, physical objects,

such as pieces of hardware or software; they are treated as black boxes exhibiting behaviour and interacting with their environment, but not amenable to formal reasoning. This makes it difficult to give a formal definition of **conforms-to** which should be our aim in a formal testing framework. In order to reason formally about implementations, we make the assumption that any real implementation IUT $\in$ *IMPS* can be modelled by a formal object $i_{\text{IUT}} \in$ *MODS*, where *MODS* is referred to as the universe of models. This assumption is referred to as the *test hypothesis* [Ber91]. Note that the test hypothesis only assumes that a model i_{IUT} exists, but not that it is known a priori.

Thus the test hypothesis allows to reason about implementations as if they were formal objects, and, consequently, to express conformance by a formal relation between models of implementations and specifications. Such a relation is called an *implementation relation* **imp** $\subseteq$ *MODS* $\times$ *SPECS* [BAL$^+$90, ISO96]. Implementation IUT $\in$ *IMPS* is said to be correct with respect to $s \in$ *SPECS*, IUT **conforms-to** s, if and only if the model $i_{\text{IUT}} \in$ *MODS* of IUT is **imp**-related to s: i_{IUT} **imp** s.

Observation and testing The behaviour of an implementation under test is investigated by performing experiments on the implementation and observing the reactions that the implementation produces to these experiments (Figure 2.3). The specification of such an experiment is called a *test case*, and the process of applying a test to an implementation under test is called *test execution*.

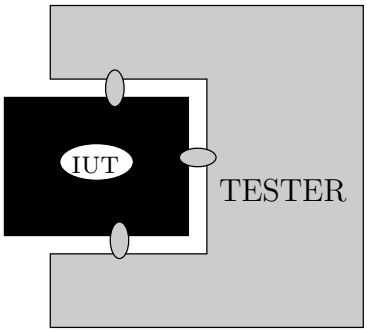


Figure 2.3: Black-box IUT and the tester

Let test cases be formally expressed as elements of a domain *TESTS*. Then test execution requires an operational procedure to execute and apply a test case $t \in$ *TESTS* to an implementation under test IUT $\in$ *IMPS*. This operational procedure is denoted by EXEC$(t,$ IUT$)$. During test execution a number of observations will be made, e.g., occurring events will be logged, or the response of the implementation to a particular stimulus will be recorded. Let (the formal interpretation of) these observations be given in a domain of observations *OBS*, then test execution EXEC$(t,$ IUT$)$ will lead to a subset of *OBS*. Note that EXEC is not a formal concept; it captures the action of "pushing the button" to let t run with IUT. Also note that EXEC$(t,$ IUT$)$ may involve multiple runs of t and IUT, e.g., in case nondeterminism is involved.

Again, since EXEC$(t,$ IUT$)$ corresponds to the physical execution of a test case, we have to model this process of test execution in our formal domain to allow formal reasoning about it. This is done by introducing an observation function obs : *TESTS* $\times$ *MODS* $\rightarrow$

$\mathcal{P}(OBS)$. So, $obs(t, i_{\text{IUT}})$ formally models the real test execution $\text{EXEC}(t, \text{IUT})$.

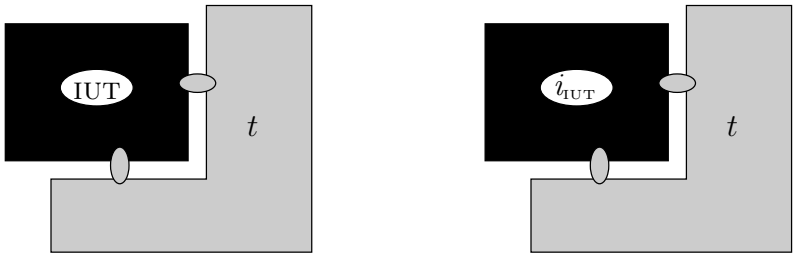


Figure 2.4: Test hypothesis: no test t can distinguish between IUT and i_{IUT}

In the context of an *observational framework* consisting of *TESTS*, *OBS*, EXEC and *obs*, it can now be stated more precisely what is meant by the test hypothesis:

$$\forall \text{IUT} \in \mathit{IMPS}\ \exists i_{\text{IUT}} \in \mathit{MODS}\ \forall t \in \mathit{TESTS}:\ \text{EXEC}(t, \text{IUT}) = obs(t, i_{\text{IUT}}) \tag{2.1}$$

This could be paraphrased as follows: for all real implementations that we are testing, it is assumed that there is a model, such that if we would put the IUT and the model in black boxes and would perform all possible experiments defined in *TESTS*, then we would not be able to distinguish between the real IUT and the model (see Figure 2.4). Actually, this notion of testing is analogous to the ideas underlying testing equivalences [DNH84, DN87], which will be elaborated for transition systems in Section 4.3.

Usually, we like to interpret observations of test execution in terms of being right or wrong. So we introduce a family of *verdict functions* $\nu_t : \mathcal{P}(OBS) \to \{\mathbf{fail}, \mathbf{pass}\}$ which allows to introduce the following abbreviation:

$$\text{IUT } \mathbf{passes}\ t \quad =_{\text{def}} \quad \nu_t(\text{EXEC}(t, \text{IUT})) = \mathbf{pass} \tag{2.2}$$

This is easily extended to a *test suite* $T \subseteq \mathit{TESTS}$: IUT **passes** $T \Leftrightarrow \forall t \in T$: IUT **passes** t. Moreover, an implementation fails test suite T if it does not pass: IUT **fails** $T \Leftrightarrow$ IUT **passes** T.

Conformance testing Conformance testing involves assessing, by means of testing, whether an implementation conforms, with respect to implementation relation **imp**, to its specification. Hence, the notions of conformance, expressed by **imp**, and of test execution, expressed by EXEC, have to be linked in such a way that from test execution an indication about conformance is obtained. So, ideally, we would like to have a test suite T_s such that for a given specification s

$$\text{IUT } \mathbf{conforms\text{-}to}\ s \iff \text{IUT } \mathbf{passes}\ T_s \tag{2.3}$$

A test suite with this property is called *complete*; it can distinguish exactly between all conforming and non-conforming implementations. Unfortunately, this is a very strong requirement for practical testing: complete test suites are usually infinite, and consequently not practically executable. Hence, usually a weaker requirement on test suites is posed: they should be *sound*, which means that all correct implementations, and possibly some incorrect implementations, will pass them; or, in other words, any detected

erroneous implementation is indeed non-conforming, but not the other way around. Soundness corresponds to the left-to-right implication in (2.3). The right-to-left implication is called *exhaustiveness*; it means that all non-conforming implementations will be detected.

To show soundness (or exhaustiveness) for a particular test suite we have to use the formal models of implementations and test execution:

$$\forall i \in \mathit{MODS}: \quad i \ \mathbf{imp}\ s \iff \forall t \in T:\ \nu_t(\mathit{obs}(t,i)) = \mathbf{pass} \tag{2.4}$$

Once (2.4) has been shown it follows that

$$\begin{array}{ll} & \text{IUT } \mathbf{passes}\ T \\ \text{iff} & (*\ \text{definition } \mathbf{passes}\ T\ *) \\ & \forall t \in T:\ \text{IUT } \mathbf{passes}\ t \\ \text{iff} & (*\ \text{definition } \mathbf{passes}\ t\ *) \\ & \forall t \in T:\ \nu_t(\text{EXEC}(t, \text{IUT})) = \mathbf{pass} \\ \text{iff} & (*\ \text{test hypothesis (2.1)}\ *) \\ & \forall t \in T:\ \nu_t(\mathit{obs}(t, i_{\text{IUT}})) = \mathbf{pass} \\ \text{iff} & (*\ \text{completeness on models (2.4) applied to } i_{\text{IUT}}\ *) \\ & i_{\text{IUT}}\ \mathbf{imp}\ s \\ \text{iff} & (*\ \text{definition of conformance}\ *) \\ & \text{IUT } \mathbf{conforms\text{-}to}\ s \end{array}$$

So, if the completeness property has been proved on the level of models and if there is ground to assume that the test hypothesis holds, then conformance of an implementation with respect to its specification can be decided by means of a testing procedure.

Now, of course, an important activity is to devise algorithms which produce sound and/or complete test suites from a specification given an implementation relation. This activity is known as *test derivation*. It can be seen as a function $\mathit{der}_{\mathbf{imp}} : \mathit{SPECS} \rightarrow \mathcal{P}(\mathit{TESTS})$. Following the requirement on soundness of test suites, such a function should produce sound test suites for any specification $s \in \mathit{SPECS}$, so the test suite $\mathit{der}_{\mathbf{imp}}(s)$ should satisfy the left-to-right implication of (2.4).

Extensions Some extensions to and refinements of the formal testing framework can be made. Two of them are mentioned here. The first one concerns the *test architecture* [Tre94, ISO96]. A test architecture defines the environment in which an implementation is tested. It gives an abstract view of how the tester communicates with the IUT. Usually, an IUT is embedded in a test context, which is there when the IUT is tested, but which is not the object of testing. In order to formally reason about testing in context, the test context must be formally modelled. Sometimes, the term SUT – system under test – is then used to denote the implementation with its test context, whereas IUT is used to denote the bare implementation without its context.

The second extension is the introduction of *coverage* within the formal framework. The coverage of a test suite can be introduced by assigning to each erroneous implementation that is detected by a test suite a value and subsequently integrating all values. This can be combined with a stochastic view on erroneous implementations and a probabilistic view on test execution [Bri93, HT96].

physical ingredients:	
black-box implementation	$\text{IUT} \in \mathit{IMPS}$
execution of test case	$\text{EXEC}(t, \text{IUT})$
formal ingredients:	
specification	$s \in \mathit{SPECS}$
implementation model	$i_{\text{IUT}} \in \mathit{MODS}$
implementation relation	$\mathbf{imp} \subseteq \mathit{MODS} \times \mathit{SPECS}$
test case	$t \in \mathit{TESTS}$
observations	OBS
model of test execution	$\mathit{obs} : \mathit{TESTS} \times \mathit{MODS} \rightarrow \mathcal{P}(\mathit{OBS})$
verdict	$\nu_t : \mathcal{P}(\mathit{OBS}) \rightarrow \{\mathbf{fail}, \mathbf{pass}\}$
test derivation	$\mathit{der}_{\mathbf{imp}} : \mathit{SPECS} \rightarrow \mathcal{P}(\mathit{TESTS})$
assumptions:	
test hypothesis	some i_{IUT} models IUT $\mathit{obs}(t, i_{\text{IUT}})$ models $\text{EXEC}(t, \text{IUT})$
prove obligation:	
soundness	$\mathit{der}_{\mathbf{imp}}(s)$ is sound for any s
exhaustiveness	$\mathit{der}_{\mathbf{imp}}(s)$ is exhaustive for any s

Table 2.1: The ingredients of the formal testing framework

2.3 Conclusion

This chapter presented a formal framework for conformance testing. Table 2.1 gives an overview of its ingredients.

The next step is the instantiation of this generic framework with specific specification formalisms, implementation models, implementation relations, tests and test derivation algorithms. Then it can be proved that those algorithms are indeed sound and exhaustive. This will be done with the formalism of labelled transition systems in Chapters 3, 4 and 5.

Chapter 3

Labelled Transition Systems

3.1 Introduction

One of the formalisms studied in the realm of conformance testing is that of *labelled transition systems*. Labelled transition systems are used as the basis for describing the behaviour of processes, such as specifications, implementations and tests. It serves as a semantic model for various formal languages, e.g., ACP [BK85], CCS [Mil89], and CSP [Hoa85]. Also (most parts of) the semantics of standardized languages like LOTOS [ISO89] and SDL [CCI92], and of the modelling language PROMELA [Hol91] can be expressed in labelled transition systems.

3.2 Labelled Transition Systems

A labelled transition system is a structure consisting of states with transitions, labelled with actions, between them. The states model the system states; the labelled transitions model the actions that a system can perform.

Definition 3.2.1
A *labelled transition system* is a 4-tuple $\langle S, L, T, s_0 \rangle$ where

- S is a countable, non-empty set of *states*;
- L is a countable set of *labels*;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the *transition relation*;
- $s_0 \in S$ is the *initial state*.

□

We write $s \xrightarrow{\mu} s'$ if there is a transition labelled μ from state s to state s', i.e., $(s, \mu, s') \in T$. This is interpreted as: "when the system is in state s it may perform action μ and go to state s' ". The labels in L represent the observable actions of a system; they model the system's interactions with its environment. Internal actions are denoted by the special label $\tau \notin L$; τ is assumed to be unobservable for the system's environment.

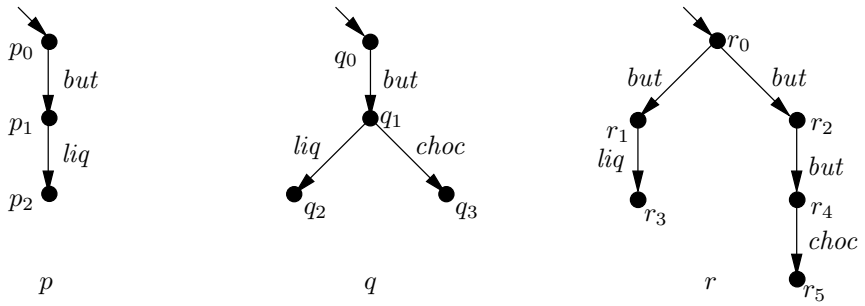


Figure 3.1: Action trees of vending machines

We can represent a labelled transition system by a tree or a graph, where nodes represent states and labelled edges represent transitions, see Figure 3.1.

Example 3.2.2
The tree q of Figure 3.1 represents the labelled transition system
$\langle\ \{q_0, q_1, q_2, q_3\},\ \{but, liq, choc\},\ \{\ \langle q_0, but, q_1\rangle,\ \langle q_1, liq, q_2\rangle,\ \langle q_1, choc, q_3\rangle\ \},\ q_0\ \rangle$.

We have that $p_0 \xrightarrow{but} p_1$. For system r we have that $r_0 \xrightarrow{but} r_1$, and also $r_0 \xrightarrow{but} r_2$. □

A *computation* is a (finite) composition of transitions:

$$s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \xrightarrow{\mu_3} \ \ldots\ \xrightarrow{\mu_{n-1}} s_{n-1} \xrightarrow{\mu_n} s_n$$

A *trace* captures the observable aspects of a computation; it is the sequence of observable actions of a computation. The set of all finite sequences of actions over L is denoted by L^*, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2.

We denote the class of all labelled transition systems over L by $\mathcal{LTS}(L)$. For technical reasons we restrict $\mathcal{LTS}(L)$ to labelled transition systems that are strongly convergent, i.e., ones that do not have infinite compositions of transitions with only internal actions.

Interactions can be concatenated using the following notation: $s \xrightarrow{a\cdot\tau\cdot c} s''$ expresses that the system, when in state s may perform the sequence of actions $a\cdot\tau\cdot c$ and end in state s''. If we abstract from the internal actions of the system we write $s \stackrel{a\cdot c}{\Longrightarrow} s''$. These and some other notations and properties are given in definition 3.2.3.

Definition 3.2.3
Let $p = \langle S, L, T, s_0\rangle$ be a labelled transition system with $s, s' \in S$, and let $\mu_{(i)} \in L \cup \{\tau\}$,

$a_{(i)} \in L$, and $\sigma \in L^*$.

$$
\begin{array}{lll}
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & =_{\text{def}} & \exists s_0, \ldots, s_n : \; s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{\text{def}} & \exists s' : \; s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} \!\!\!\!/ & =_{\text{def}} & \text{not } \exists s' : \; s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \stackrel{\epsilon}{\Longrightarrow} s' & =_{\text{def}} & s = s' \;\text{ or }\; s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \stackrel{a}{\Longrightarrow} s' & =_{\text{def}} & \exists s_1, s_2 : \; s \stackrel{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\epsilon}{\Longrightarrow} s' \\
s \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} s' & =_{\text{def}} & \exists s_0 \ldots s_n : \; s = s_0 \stackrel{a_1}{\Longrightarrow} s_1 \stackrel{a_2}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} s_n = s' \\
s \stackrel{\sigma}{\Longrightarrow} & =_{\text{def}} & \exists s' : \; s \stackrel{\sigma}{\Longrightarrow} s' \\
s \stackrel{\sigma}{\Longrightarrow} \!\!\!\!/ & =_{\text{def}} & \text{not } \exists s' : s \stackrel{\sigma}{\Longrightarrow} s'
\end{array}
$$

□

Example 3.2.4
In Figure 3.1 we have that $r_0 \xrightarrow{but \cdot liq} r_3$, so also $r_0 \xrightarrow{but \cdot liq}$, but $r_0 \xrightarrow{but \cdot choc} \!\!\!\!/$. Since there are no τ-actions in Figure 3.1 we can replace any $\xrightarrow{\sigma}$ by $\stackrel{\sigma}{\Longrightarrow}$.
□

We will not always distinguish between a transition system and its initial state: if $p = \langle S, L, T, s_0 \rangle$, we will identify the process p with its initial state s_0, e.g., we write $p \stackrel{\sigma}{\Longrightarrow}$ instead of $s_0 \stackrel{\sigma}{\Longrightarrow}$.

Definition 3.2.5

1. $init(p) \;=_{\text{def}}\; \{\, \mu \in L \cup \{\tau\} \mid p \xrightarrow{\mu} \,\}$
2. $traces(p) \;=_{\text{def}}\; \{\, \sigma \in L^* \mid p \stackrel{\sigma}{\Longrightarrow} \,\}$
3. $p \,\textbf{after}\, \sigma \;=_{\text{def}}\; \{\, p' \mid p \stackrel{\sigma}{\Longrightarrow} p' \,\}$
4. $P \,\textbf{after}\, \sigma \;=_{\text{def}}\; \bigcup\{\; p \,\textbf{after}\, \sigma \mid p \in P \,\}$ where P is a set of states.
5. $der(p) \;=_{\text{def}}\; \{\, p' \mid \exists \sigma \in L^* : p \stackrel{\sigma}{\Longrightarrow} p' \,\}$
6. p has *finite behaviour* if there is a natural number n such that all traces in $traces(p)$ have length smaller than n.
7. p is *finite state* if the number of reachable states $der(p)$ is finite.
8. p is *deterministic* if, for all $\sigma \in L^*$, $p \,\textbf{after}\, \sigma$ has at most one element. If $\sigma \in traces(p)$, then $p \,\textbf{after}\, \sigma$ is overloaded to denote this element.

□

Example 3.2.6
In Figure 3.1, $init(q) = init(q_0) = \{but\}$; $init(q_1) = \{liq, choc\}$ and $init(r) = \{but\}$.

All possible execution sequences of process r are given by
$traces(r) = \{\, \epsilon,\; but,\; but \cdot liq,\; but \cdot but,\; but \cdot but \cdot choc \,\}$.

Some states in which a system can be **after** a trace are:
$r \,\textbf{after}\, \epsilon = \{r_0\}$; $r \,\textbf{after}\, but = \{r_1, r_2\}$; $r \,\textbf{after}\, but \cdot choc = \emptyset$; and
$(\, r \,\textbf{after}\, but \,) \,\textbf{after}\, but = \{r_1, r_2\} \,\textbf{after}\, but = \{r_4\}$.

The derivatives of q are all states that can be reached from the initial state q_0: $der(q) = \{q_0, q_1, q_2, q_3\}$.

All three processes p, q and r have finite behaviour and are finite state; p and q are deterministic, while r is nondeterministic.

□

3.3 Representing Labelled Transition Systems

Labelled transition systems form a semantic model to reason over processes, such as specifications, implementations and tests. Except for the most trivial processes, like the ones in Figure 3.1, a representation by means of trees or graphs is usually not feasible. Realistic systems easily have billions of states, so that drawing them is not an option. We need other ways of representing transition systems and we use a *process algebraic language* to do that. An additional advantage of such a representation is that with expressions over such a language it is much easier to preserve structure and to compose complex transition systems from simpler ones.

There exist many process algebras. We use a variant of the language LOTOS [BB87, ISO89]. We introduce some of the constructions here, although this text is not intended as a tutorial in process algebra; we refer to the standard literature for a more detailed treatment [Hoa85, Mil89, BB87, ISO89].

Behaviour is described by *behaviour expressions*. We use the following syntax for a behaviour expression B:

$$B \quad =_{\text{def}} \quad a \; ; \; B \quad | \quad \mathbf{i} \; ; \; B \quad | \quad \Sigma \, \mathcal{B} \quad | \quad B \; |[G]| \; B \quad | \quad P$$

The *action prefix expression* $a; B$, with $a \in L$, describes the behaviour which can perform the action a and then behaves as B.

The operational semantics of a behaviour expression is usually formally defined by means of axioms and inference rules. These define for each behaviour expression, in finitely many steps, all its possible transitions. For the expression $a; B$ this is the following axiom:

$$\vdash \quad a; B \xrightarrow{a} B$$

This axiom is to be read as: an expression of the form $a; B$ can always make a transition $\xrightarrow{a}$ to a state from where it behaves as B.

The expression $\mathbf{i}; B$ is analogous to $a; B$, the difference being that $\mathbf{i}$ denotes an internal action τ in the transition system:

$$\vdash \quad \mathbf{i}; B \xrightarrow{\tau} B$$

The *choice expression* $\Sigma \, \mathcal{B}$, where $\mathcal{B}$ is a countable set of behaviour expressions, denotes a choice of behaviour. It behaves as any of the processes in the set $\mathcal{B}$. It is formally defined by the inference rule:

$$B \xrightarrow{\mu} B', \; B \in \mathcal{B}, \; \mu \in L \cup \{\tau\} \quad \vdash \quad \Sigma \, \mathcal{B} \xrightarrow{\mu} B'$$

This inference rule to be read as follows: suppose that we know that B can make a transition to B'; moreover we have that $B \in \mathcal{B}$ and μ is any observable or internal action, then we can conclude that $\Sigma \, \mathcal{B}$ can make the same transition to B'.

We use $B_1 \; \Box \; B_2$ as an abbreviation of $\Sigma \, \{B_1, B_2\}$, i.e., $B_1 \; \Box \; B_2$ behaves as either B_1 or B_2. The expression **stop** is an abbreviation for $\Sigma \, \emptyset$, i.e., it is the behaviour which cannot perform any action, so it is the deadlocked process.

The *parallel expression* $B_1 |[G]| B_2$, where $G \subseteq L$, denotes the parallel execution of B_1 an B_2. In this parallel execution all actions in G must synchronize, while all actions

not in G (including τ) can occur independently in both processes, i.e., *interleaved*. We use $||$ as an abbreviation for $|[L]|$, i.e., sychronization on all actions except τ, and $|||$ as an abbreviation for $|[\emptyset]|$, i.e., full interleaving and no synchronization. The inference rules are as follows:

$$\begin{array}{lcl} B_1 \xrightarrow{\mu} B_1',\ \mu \in (L \cup \{\tau\}) \backslash G & \vdash & B_1|[G]|B_2 \xrightarrow{\mu} B_1'|[G]|B_2 \\ B_2 \xrightarrow{\mu} B_2',\ \mu \in (L \cup \{\tau\}) \backslash G & \vdash & B_1|[G]|B_2 \xrightarrow{\mu} B_1|[G]|B_2' \\ B_1 \xrightarrow{a} B_1',\ B_2 \xrightarrow{a} B_2',\ a \in G & \vdash & B_1|[G]|B_2 \xrightarrow{a} B_1'|[G]|B_2' \end{array}$$

A *process definition* links a process name to a behaviour expression:

$$P \ := \ B_P$$

The name P can then be used in behaviour expressions to stand for the behaviour expressed by its corresponding behaviour expression. Formally:

$$B_P \xrightarrow{\mu} B',\ P := B_P,\ \mu \in L \cup \{\tau\} \quad \vdash \quad P \xrightarrow{\mu} B'$$

As usual, parentheses are used to disambiguate expressions. If no parentheses are used ';' binds stronger than '$\Box$', which binds stronger than '$|[G]|$'. The parallel operators read from left to right – they are not associative for different synchronization sets. So, $a; B_1 \ || \ b \ ; B_2 \ \Box \ c; B_3 \ ||| \ d; B_4$ is read as $(((a; B_1) \ || \ ((b \ ; B_2) \ \Box \ (c; B_3))) \ ||| \ (d; B_4))$.

Note that not every behaviour expression represents a transition system in $\mathcal{LTS}(L)$, e.g., the transition system defined by $P := \mathbf{i}; P$ is not strongly convergent, and thus is not in $\mathcal{LTS}(L)$.

Example 3.3.1
Behaviour expressions representing the processes of Figure 3.1 are:

$$\begin{array}{lcl} p & : & \mathit{but};\ \mathit{liq};\ \mathbf{stop} \\ q & : & \mathit{but};\ (\ \mathit{liq};\ \mathbf{stop} \ \Box \ \mathit{choc};\ \mathbf{stop}\ ) \\ r & : & \mathit{but};\ \mathit{liq};\ \mathbf{stop} \ \Box \ \mathit{but};\ \mathit{but};\ \mathit{choc};\ \mathbf{stop} \end{array}$$

These behaviour expressions are not unique, e.g., we could also choose

$$\begin{array}{lcl} p & : & \mathit{but};\ \mathit{liq};\ \mathit{liq};\ \mathbf{stop} \ || \ \mathit{but};\ \mathit{liq};\ \mathit{choc};\ \mathbf{stop} \\ q & : & \mathit{but};\ \Sigma\{\mathit{liq};\ \mathbf{stop},\ \mathit{choc};\ \mathbf{stop}\} \end{array}$$

In particular, the parallel operator can be used to efficiently represent large transition systems. Consider the process p of Figure 3.1 which has 3 states. The interleaving of p with itself, $p|||p$, has $3 \times 3 = 9$ states. And $p|||p|||p$ has 27 states, and so forth.

If $W \ := \ a; b; W$ then W is the transition system consisting of two states endlessly performing the actions a and b alternatingly. The process $V \ := \ a; V \ \Box \ b; V$ also performs a and b infinitely often, but not necessarily alternatingly.

Both V and W have infinite behaviour, but they are finite state. (cf. Definition 3.2.5). An example of an infinite state process is $U \ := \ a \ ; \ (\ b \ ; \ \mathbf{stop} \ ||| \ U \ )$. U can perform actions a and b but it can never do more b's than a's.

□

3.4 Input-Output Transition Systems

With labelled transition systems, communication between a process and its environment – also modelled as a process – is based on symmetric interaction, as expressed

by the parallel operator $|[\ldots]|$. An interaction can occur if both the process and its environment are able to perform that interaction, implying that they can also both block the occurrence of an interaction. If both offer more than one interaction then it is assumed that by some mysterious negotiation mechanism they will agree on a common interaction. There is no notion of input or output, nor of initiative or direction. All actions are treated in the same way for both communicating partners.

Many real systems, however, communicate in a different manner. They do make a distinction between inputs and outputs, and one can clearly distinguish whether the initiative for a particular interaction is with the system or with its environment. There is a direction in the flow of information from the initiating communicating process to the other. The initiating process determines which interaction will take place. Even if the other one decides not to accept the interaction, this is usually implemented by first accepting it, and then initiating a new interaction in the opposite direction explicitly signaling the non-acceptance. One could say that the mysterious negotiation mechanism is made explicit by exchanging two messages: one to propose an interaction and a next one to inform the initiating process about the (non-)acceptance of the proposed interaction.

We use *input-output transition systems*, analogous to Input/Output Automata [LT89], to model systems for which the set of actions can be partitioned into *output actions*, for which the initiative to perform them is with the system, and *input actions*, for which the initiative is with the environment. If an input action is initiated by the environment, the system is always prepared to participate in such an interaction: all the inputs of a system are always enabled; they can never be refused. Naturally an input action of the system can only interact with an output of the environment, and vice versa, implying that output actions can never be blocked by the environment. Although the initiative for any interaction is in exactly one of the communicating processes, the communication is still synchronous: if an interaction occurs, it occurs at exactly the same time in both processes. The communication, however, is not symmetric: the communicating processes have different roles in an interaction.

Definition 3.4.1
An *input-output transition system* p is a labelled transition system in which the set of actions L is partitioned into input actions L_I and output actions L_U ($L_I \cup L_U = L,\ L_I \cap L_U = \emptyset$), and for which all input actions are always enabled in any state:

$$\text{whenever} \quad p \stackrel{\sigma}{\Longrightarrow} p' \quad \text{then} \quad \forall a \in L_I :\ p' \stackrel{a}{\Longrightarrow}$$

The class of input-output transition systems with input actions in L_I and output actions in L_U is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I \cup L_U)$.

□

Example 3.4.2
Figure 3.2 gives some input-output transition systems with $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$. In q_1 we can push the *button*, which is an input for the candy machine, and then the machine outputs *liquorice*. After the *button* has been pushed once, and also after the machine has released *liquorice*, any more pushing of the *button* has no effect: the machine makes a self-loop. In this text we use the convention that a self-loop of a state that is not explicitly labelled is labelled with all inputs that cannot occur in that state (and also not via τ-transitions, cf. definition 3.4.1).

□

Since input-output transition systems are labelled transition systems, all definitions for

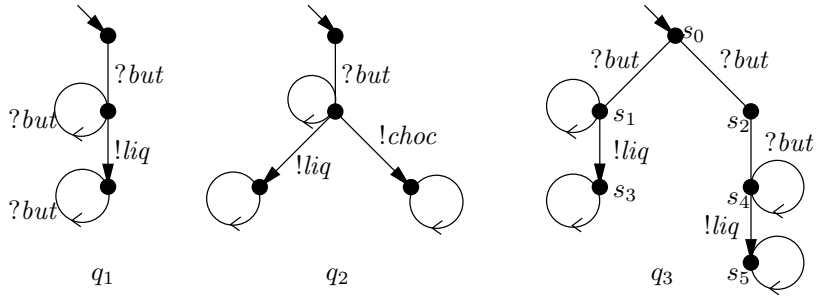


Figure 3.2: Input-output transition systems

labelled transition systems apply. In particular, synchronous parallel communication can be expressed by $|[\ldots]|$, but now care should be taken that the outputs of one process interact with the inputs of the other.

An input-output system can autonomously decide to perform an output action; the environment will always accept it. So, in states where output actions are possible the system itself can decide whether to continue or not. On the other hand, in states where no output actions are enabled the system has to wait for its environment to supply the next input action. We say that the system is *suspended* or *quiescent*. A state where no output actions are enabled is called a *quiescent state* and a trace which can lead to such a state is called a *quiescent trace*.

Definition 3.4.3
Let $p \in \mathcal{LTS}(L_I \cup L_U)$.

1. A state p is *quiescent*, denoted by $\delta(p)$, if $\forall \mu \in L_U \cup \{\tau\} : p \stackrel{\mu}{\not\longrightarrow}$
2. A *quiescent trace* of p is a trace σ that may lead to a quiescent state: $\exists p' \in (\, p \textbf{ after } \sigma \,) : \delta(p)$
3. The set of quiescent traces of p is denoted by $Qtraces(p)$.

□

It turns out that we can conveniently express quiescence as a 'normal' output action. We denote this 'action' by δ ($\delta \notin L \cup \{\tau\}$). We can then consider transitions of the form $p \stackrel{\delta}{\longrightarrow} p'$ and traces like $p \stackrel{a \cdot b \cdot \delta \cdot c}{\Longrightarrow}$. If p can 'perform' a δ-transition then this is interpreted as p being quiescent, i.e., p cannot perform any output action. In this way the absence of outputs, i.e., quiescence, is made into an explicitly observable event. Since $p \stackrel{\delta}{\longrightarrow} p'$ does not correspond to any 'real' transition being performed the resulting state after a δ-transition is always the same as the starting state, $p = p'$, so: $p \stackrel{\delta}{\longrightarrow} p$, and we have

$$\delta(p) \quad \text{iff} \quad p \stackrel{\delta}{\longrightarrow} p$$

Concretely, one could think of 'observing' δ in a system state if after any waiting period still no outputs are produced by the system.

Suspension traces are introduced as traces in which the special action δ may occur. For example, the trace $\delta \cdot ?a \cdot \delta \cdot ?b \cdot !x$ expresses that initially no outputs can be observed; then, after input action $?a$, there is again no output; and then, after input $?b$ is performed,

the output $!x$ can be observed. We write L_δ for $L \cup \{\delta\}$, so suspension traces are traces in L_δ^*.

Definition 3.4.4
Let the transition relation $\xrightarrow{\mu}$ be extended with quiescence transitions $p \xrightarrow{\delta} p$. Then the *suspension traces* of process $p \in \mathcal{LTS}(L)$ are:

$$Straces(p) \quad =_{\text{def}} \quad \{\ \sigma \in L_\delta^* \mid p \stackrel{\sigma}{\Longrightarrow}\ \}$$

□

For input-output transition systems all output actions that are enabled in state p are collected into the set $out(p)$. Moreover, $out(p)$ contains an indication whether p is quiescent: if $\delta(p)$ then δ is added to $out(p)$.

Definition 3.4.5
Let p be a state in a transition system, and let P be a set of states, then

1. $out(p) \quad =_{\text{def}} \quad \{\ x \in L_U \mid p \xrightarrow{x}\ \} \ \cup\ \{\ \delta \mid \delta(p)\ \}$
2. $out(P) \quad =_{\text{def}} \quad \bigcup \{\ out(p) \mid p \in P\ \}$

□

In particular, this definition will be used in combination with $.\,\mathbf{after}\,.$ (cf. Definition 3.2.5); $out(\,p\,\mathbf{after}\,\sigma\,)$ gives all possible output actions, including quiescence, which process p may produce after having performed the suspension trace σ.

Example 3.4.6
In Figure 3.2, we have that for q_1 the initial state and the final state are quiescent; the traces ϵ and $?but{\cdot}!liq$ are quiescent traces of q_1, but also $?but{\cdot}!liq{\cdot}\delta$ is a quiescent trace.

For q_3 the states s_0, s_2, s_3 and s_5 are quiescent, so, for example, we can write $s_2 \xrightarrow{\delta} s_2$. The suspension traces are $Straces(q_3) = \{\epsilon,\ \delta,\ \delta{\cdot}\delta,\ \ldots,\ ?but{\cdot}!liq,\ ?but{\cdot}!liq{\cdot}\delta,\ ?but{\cdot}!liq{\cdot}\delta{\cdot}\delta,\ \ldots,\ \delta{\cdot}\delta{\cdot}?but{\cdot}!liq{\cdot}\delta{\cdot}\delta,\ \ldots,\ ?but{\cdot}?but{\cdot}!liq,\ ?but{\cdot}\delta{\cdot}?but{\cdot}!liq,\ ?but{\cdot}\delta{\cdot}\delta{\cdot}?but{\cdot}!liq,\ \ldots,\ ?but{\cdot}\delta{\cdot}\delta{\cdot}?but{\cdot}!liq{\cdot}\delta,\ \ldots\}$.

Some out-sets for q_1:

$$\begin{array}{lcl}
out(\,q_1\ \mathbf{after}\ \epsilon\,) & = & \{\delta\} \\
out(\,q_1\ \mathbf{after}\ ?but\,) & = & \{!liq\} \\
out(\,q_1\ \mathbf{after}\ ?but{\cdot}?but\,) & = & \{!liq\} \\
out(\,q_1\ \mathbf{after}\ !liq\,) & = & \emptyset \\
out(\,q_1\ \mathbf{after}\ \delta\,) & = & \{\delta\} \\
out(\,q_1\ \mathbf{after}\ ?but{\cdot}!liq\,) & = & \{\delta\} \\
out(\,q_1\ \mathbf{after}\ ?but{\cdot}!liq{\cdot}?but\,) & = & \{\delta\} \\
out(\,q_1\ \mathbf{after}\ ?but{\cdot}!liq{\cdot}!liq\,) & = & \emptyset
\end{array}$$

And for q_2:

$$out(\,q_1\ \mathbf{after}\ ?but\,) \quad = \quad \{!liq, !choc\}$$

And some out-sets for q_3:

$$
\begin{array}{lcl}
out(s_0) & = & \{\delta\} \\
out(s_1) & = & \{!liq\} \\
out(s_2) & = & \{\delta\} \\
out(\, q_3 \textbf{ after } ?but\,) & = & \{!liq, \delta\} \\
out(\, q_3 \textbf{ after } ?but{\cdot}?but\,) & = & out(s_1) \cup out(s_4) \;=\; \{!liq\} \\
out(\, q_3 \textbf{ after } ?but{\cdot}\delta{\cdot}?but\,) & = & out(s_4) \;=\; \{!liq\} \\
out(\, q_3 \textbf{ after } ?but{\cdot}?but{\cdot}!liq\,) & = & \{\delta\}
\end{array}
$$

□

3.5 Transition Systems in the Testing Framework

Labelled transition systems are chosen as the basis for instantiating the generic, formal testing framework of Chapter 2. This means that the formal domains *SPECS*, *MODS* and *TESTS* will now consist of (some kind of) transition systems. For *SPECS* and *MODS* this instantiation is given in this section; for *TESTS* this will be done in Chapter 5.

Specifications For specifications we allow to use labelled transition systems, or any formal language with a labelled transition system semantics. We require that the actions of the transition system are known and can be partitioned into inputs and outputs, denoted by L_I and L_U, respectively. However, we do not impose any restrictions on inputs or outputs. So, *SPECS* is instantiated with $\mathcal{LTS}(L_I \cup L_U)$ (definition 3.2.1).

Implementations and their models We assume that implementations can be modelled as labelled transition systems over the same inputs L_I and outputs L_U as the specification. Moreover, we assume that implementations can always perform all their inputs, i.e., any input action is always enabled in any state. So, *MODS* is instantiated with $\mathcal{IOTS}(L_I, L_U)$ (definition 3.4.1).

For *IMPS* we allow any computer system or program which can be modelled as an input-output transition system, i.e., a system which has distinct inputs and outputs, where inputs can be mapped 1:1 on L_I and outputs on L_U, and where inputs can always occur.

Chapter 4

Relating Transition Systems

4.1 Introduction

Labelled transition systems are used to model the observable behaviour of systems, such as distributed systems and protocols. As such they are a suitable formalization of the notion of a process. However, 'some behaviours are more equal than others': it may occur that different labelled transition systems intuitively describe the same observable behaviour, e.g., when an action τ occurs, which is assumed to be unobservable, or when a state has two equally labelled outgoing transitions to the same state. Therefore equivalence relations have been defined based on the notion of observable behaviour, and equality of behaviour is studied with respect to equivalence classes. A lot of equivalence relations are known from literature: observation equivalence [Mil80], strong and weak bisimulation equivalence [Par81, Mil89], failure equivalence [Hoa85], testing equivalence [DNH84], failure trace equivalence [Bae86], generalized failure equivalence [Lan90], and many others; for an overview see [Gla90, Gla93].

Sometimes an equivalence is too strong to compare transition systems and other relations than equivalences are considered. Such relations are usually intended to express the notion that one system implements or refines another one. They can be used as implementation relations, cf. Section 2.2. Naturally, such relations are often *preorders*, i.e., relations which are reflexive (any system implements itself) and transitive (a correct implementation of a correct implementation is a correct implementation). A preorder $\sqsubseteq$ can always be related to a corresponding equivalence $\approx$: $\approx \;=_{\text{def}}\; \sqsubseteq \cap \sqsupseteq$, where $\sqsupseteq \;=_{\text{def}}\; \sqsubseteq^{-1}$.

This chapter discusses some equivalences, some preorders and some non-preorder implementation relations. The most important one for testing is **ioco**, which will be used as implementation relation in subsequent chapters. It will be discussed in Section 4.5. The main part of that section is intended to be self-contained. Sections 4.2, 4.3 and 4.4 discuss some other interesting relations, including those which can be considered as **ioco**'s ancestors, but these sections are not a prerequisite for subsequent chapters.

4.2 Equality

In comparing labelled transition systems the first observation is that the names of states, nor the existence of states that cannot be reached during any execution, influence the observable behaviour. Two labelled transition systems are *isomorphic* if their reachable states can be mapped one-to-one to each other, preserving transitions and initial states. In fact we already used this with giving the different behaviour expressions for p and q in example 3.3.1.

The second observation is that equally labelled transitions to states with equivalent behaviour cannot be discerned. *Strong bisimulation equivalence* $\sim$ declares such systems equivalent.

Weak bisimulation equivalence $\approx$ requires a relation between the reachable states of two systems that can simulate each other: if one can perform a trace σ, the other must be able to do the same, and vice versa, and the resulting states must simulate each other again.

If the *traces* of two systems are equal they are called *trace equivalent* $\approx_{tr}$.

Definition 4.2.1
Let $p_1, p_2 \in \mathcal{LTS}$, $p_1 = \langle S_1, L, T_1, s_{0_1}\rangle$, $p_2 = \langle S_2, L, T_2, s_{0_2}\rangle$:

1. isomorphism:
   $p_1 \equiv p_2 \quad =_{\text{def}} \quad$ there exists a bijection $f : der(p_1) \to der(p_2)$
   such that $\forall s_1, s_2 \in der(p_1)$, $\mu \in L \cup \{\tau\}$:
   $s_1 \xrightarrow{\mu} s_2$ iff $f(s_1) \xrightarrow{\mu} f(s_2)$
   and $f(s_{0_1}) = s_{0_2}$
2. strong bisimulation:
   $p_1 \sim p_2 \quad =_{\text{def}} \quad \exists \mathcal{R} \subseteq der(p_1) \times der(p_2), \text{such that} \langle s_{0_1}, s_{0_2}\rangle \in \mathcal{R},$ and
   $\forall \langle s_1, s_2\rangle \in \mathcal{R},\ \forall \mu \in L \cup \{\tau\}:$
   $\forall s_1' :$ if $s_1 \xrightarrow{\mu} s_1'$ then $\exists s_2' : s_2 \xrightarrow{\mu} s_2'$ and $\langle s_1', s_2'\rangle \in \mathcal{R}$; and
   $\forall s_2' :$ if $s_2 \xrightarrow{\mu} s_2'$ then $\exists s_1' : s_1 \xrightarrow{\mu} s_1'$ and $\langle s_1', s_2'\rangle \in \mathcal{R}$
3. weak bisimulation:
   $p_1 \approx p_2 \quad =_{\text{def}} \quad \exists \mathcal{R} \subseteq der(p_1) \times der(p_2), \text{such that} \langle s_{0_1}, s_{0_2}\rangle \in \mathcal{R},$ and
   $\forall \langle s_1, s_2\rangle \in \mathcal{R},\ \forall \sigma \in L^* :$
   $\forall s_1' :$ if $s_1 \stackrel{\sigma}{\Longrightarrow} s_1'$ then $\exists s_2' : s_2 \stackrel{\sigma}{\Longrightarrow} s_2'$ and $\langle s_1', s_2'\rangle \in \mathcal{R}$; and
   $\forall s_2' :$ if $s_2 \stackrel{\sigma}{\Longrightarrow} s_2'$ then $\exists s_1' : s_1 \stackrel{\sigma}{\Longrightarrow} s_1'$ and $\langle s_1', s_2'\rangle \in \mathcal{R}$
4. trace equivalence:
   $p_1 \approx_{tr} p_2 \quad =_{\text{def}} \quad traces(p_1) = traces(p_2)$

□

Proposition 4.2.2

1. $\equiv$, $\sim$, $\approx$, and $\approx_{tr}$ are equivalences.
2. $\equiv \ \subset \ \sim \ \subset \ \approx \ \subset \ \approx_{tr}$

□

Also non-equivalence relations over labelled transition systems can be defined. Using the concepts introduced until now two prominent ones are *trace preorder* $\leq_{tr}$ and its inverse $\geq_{tr}$. The intuition behind $\leq_{tr}$ as an implementation relation is that an

implementation $i \in \mathcal{LTS}(L)$ may show only behaviour (in terms of traces of observable actions) which is specified in the specification $s \in \mathcal{LTS}(L)$.

Definition 4.2.3
Let $p_1, p_2 \in \mathcal{LTS}$:

1. $p_1 \leq_{tr} p_2 \quad =_{\text{def}} \quad traces(p_1) \subseteq traces(p_2)$
2. $p_1 \geq_{tr} p_2 \quad =_{\text{def}} \quad traces(p_1) \supseteq traces(p_2)$

□

Proposition 4.2.4

1. $\approx_{tr} \;=\; \leq_{tr} \;\cap\; \geq_{tr}$
2. $\leq_{tr}$ and $\geq_{tr}$ are preorders

□

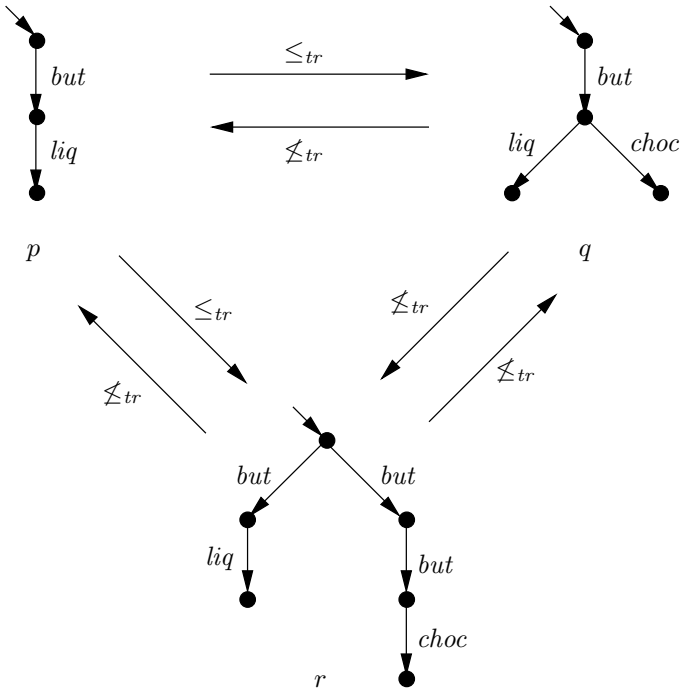


Figure 4.1:

Example 4.2.5
Figure 4.1 illustrates the trace-preorder relation $\leq_{tr}$ (and indirectly $\geq_{tr}$) for the processes of Figure 3.1. □

4.3 Relations based on Symmetric Interaction

This section continues the discussion on implementation relations for labelled transition systems. Many of these relations can be defined in an extensional way, which means that they are defined by explicitly comparing an implementation with a specification in terms of comparing the observations that an external observer can make [DNH84, DN87]. The intuition is that an implementation i is related to a specification s if any observation that can be made of i in any possible environment u can be related to, or explained from, an observation of s in the same environment u, see Figure 4.2:

$$i \textbf{ imp } s \quad =_{\text{def}} \quad \forall u \in \mathcal{U} : \ OBS(u,i) \ * \ OBS(u,s) \tag{4.1}$$

By varying the class of external observers $\mathcal{U}$, the observations OBS that an observer can make of i and s, and the relation $*$ between observations of i and s, many different implementation relations can be defined.

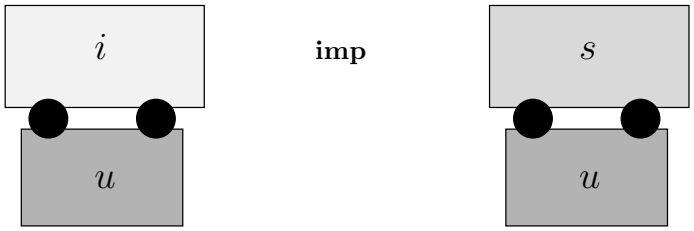


Figure 4.2: i is related to s if and only if any possible environment u can make the same observations from i as from s

One of the relations that can be expressed following (4.1) is *testing preorder* $\leq_{te}$, which we formalize in a slightly different setting from the one in [DNH84, DN87]. It is obtained if labelled transition systems are chosen as observers $\mathcal{U}$, the relation between observations is set inclusion, and the observations are traces. These traces are obtained from computations of i or s, in parallel with an observer u, where a distinction is made between normal traces and completed traces, i.e., traces which correspond to a computation after which no more actions are possible.

Definition 4.3.1
Let $p, i, s \in \mathcal{LTS}(L)$, $\sigma \in L^*$, and $A \subseteq L$, then

1. $p \textbf{ after } \sigma \textbf{ refuses } A \quad =_{\text{def}} \quad \exists p' : \ p \stackrel{\sigma}{\Longrightarrow} p' \ \text{ and } \ \forall a \in A : \ p' \stackrel{a}{\not\Longrightarrow}$
2. $p \textbf{ after } \sigma \textbf{ deadlocks} \quad =_{\text{def}} \quad p \textbf{ after } \sigma \textbf{ refuses } L$
3. The sets of *observations* obs_c and obs_t that an observer $u \in \mathcal{LTS}(L)$ can make of process $p \in \mathcal{LTS}(L)$ are given by the completed traces and the traces, respectively, of their synchronized parallel communication $u \parallel p$:

$$\begin{array}{lll} obs_c(u,p) & =_{\text{def}} & \{\ \sigma \in L^* \mid \ (u \parallel p) \textbf{ after } \sigma \textbf{ deadlocks} \ \} \\ obs_t(u,p) & =_{\text{def}} & \{\ \sigma \in L^* \mid \ (u \parallel p) \stackrel{\sigma}{\Longrightarrow} \ \} \end{array}$$

4. $i \leq_{te} s \quad =_{\text{def}} \quad \forall u \in \mathcal{LTS}(L) : obs_c(u,i) \subseteq obs_c(u,s)$ and $obs_t(u,i) \subseteq obs_t(u,s)$
5. $\approx_{te} \ = \ \leq_{te} \cap GEPRE$

□

The definitions in 4.3.1 are based on the occurrence, or absence, of observable actions. It is straightforward to show that on our class of strongly convergent transition systems these definitions correspond to those sometimes found in the literature, which also take internal actions into account:

$$p \textbf{ after } \sigma \textbf{ refuses } A \quad \text{iff} \quad \exists p' : \; p \stackrel{\sigma}{\Longrightarrow} p' \;\text{ and }\; \forall \mu \in A \cup \{\tau\} : \; p' \stackrel{\mu}{\not\longrightarrow} \tag{4.2}$$

The extensional definition of $\leq_{te}$ in definition 4.3.1 can be rewritten into an intensional characterization, i.e., a characterization in terms of properties of the labelled transition systems themselves. This characterization, given in terms of failure pairs, is known to coincide with failure preorder for strongly convergent transition systems [DN87, Tre92].

Proposition 4.3.2
$i \leq_{te} s$ iff $\forall \sigma \in L^*, \; \forall A \subseteq L : \;$ i **after** σ **refuses** A implies s **after** σ **refuses** A

□

A weaker implementation relation that is strongly related to $\leq_{te}$ is the relation **conf** [BSS87]. It is a modification of $\leq_{te}$ by restricting all observations to only those traces that are contained in the specification s. This restriction is in particular used in conformance testing. It makes testing a lot easier: only traces of the specification have to be considered, not the huge complement of this set, i.e., the traces not explicitly specified. In other words, **conf** requires that an implementation does what it should do, not that it does not do what it is not allowed to do. So a specification only partially prescribes the required behaviour of the implementation. Several test generation algorithms have been developed for the relation **conf** [Bri88, PF90, Wez90, Tre92].

Definition 4.3.3

$$\begin{aligned} i \textbf{ conf } s \quad =_{\text{def}} \quad & \forall u \in \mathcal{LTS}(L) : \quad (obs_c(u,i) \cap traces(s)) \subseteq obs_c(u,s) \\ & \text{and} \quad (obs_t(u,i) \cap traces(s)) \subseteq obs_t(u,s) \end{aligned}$$

□

Proposition 4.3.4
i **conf** s iff $\forall \sigma \in traces(s), \forall A \subseteq L : \;$ i **after** σ **refuses** A implies s **after** σ **refuses** A

□

A relation with more discriminating power than testing preorder is obtained, following (4.1), by having more powerful observers that can detect not only the occurrence of actions but also the absence of actions, i.e., refusals [Phi87]. We follow [Lan90] in modelling the observation of a refusal by adding a special label $\theta \notin L$ to observers: $\mathcal{U} = \mathcal{LTS}(L_\theta)$, where we write L_θ for $L \cup \{\theta\}$. While observing a process, a transition labelled with θ can only occur if no other transition is possible. In this way the observer knows that the process under observation cannot perform the other actions it offers. A parallel synchronization operator $\rceil\!\!\lceil$ is introduced, which models the communication between an observer with θ-transitions and a normal process, i.e., a transition system without θ-transitions. The implementation relation defined in this way is called *refusal preorder* $\leq_{rf}$.

Definition 4.3.5

1. The operator $\rceil\!\!\lceil : \mathcal{LTS}(L_\theta) \times \mathcal{LTS}(L) \to \mathcal{LTS}(L_\theta)$ is defined by the following

inference rules:

$$\begin{array}{ll}
u \xrightarrow{\tau} u' & \vdash \ u\rceil\!|\, p \xrightarrow{\tau} u'\rceil\!|\, p \\
p \xrightarrow{\tau} p' & \vdash \ u\rceil\!|\, p \xrightarrow{\tau} u\rceil\!|\, p' \\
u \xrightarrow{a} u',\ p \xrightarrow{a} p',\ a \in L & \vdash \ u\rceil\!|\, p \xrightarrow{a} u'\rceil\!|\, p' \\
u \xrightarrow{\theta} u',\ u \xrightarrow{\tau}\!\!\!\!\!/\ ,\ p \xrightarrow{\tau}\!\!\!\!\!/\ ,\ \forall a \in L:\ u \xrightarrow{a}\!\!\!\!\!/\ \text{ or } \ p \xrightarrow{a}\!\!\!\!\!/\ & \vdash \ u\rceil\!|\, p \xrightarrow{\theta} u'\rceil\!|\, p
\end{array}$$

2. The sets of *observations* obs_c^θ and obs_t^θ that an observer $u \in \mathcal{LTS}(L_\theta)$ can make of process $p \in \mathcal{LTS}(L)$ are given by the completed traces and the traces, respectively, of the synchronized parallel communication $\rceil\!|$ of u and p:

$$\begin{array}{lll}
obs_c^\theta(u,p) & =_{\text{def}} & \{\ \sigma \in L_\theta^* \ \mid\ (u\rceil\!|\, p) \textbf{ after } \sigma \textbf{ deadlocks}\ \} \\
obs_t^\theta(u,p) & =_{\text{def}} & \{\ \sigma \in L_\theta^* \ \mid\ (u\rceil\!|\, p) \stackrel{\sigma}{\Longrightarrow}\ \}
\end{array}$$

3. $i \leq_{rf} s \ =_{\text{def}} \ \forall u \in \mathcal{LTS}(L_\theta) : obs_c^\theta(u,i) \subseteq obs_c^\theta(u,s)$ and $obs_t^\theta(u,i) \subseteq obs_t^\theta(u,s)$

□

A corresponding intensional characterization of refusal preorder can be given in terms of failure traces [Gla90, Lan90]. A failure trace is a trace in which both actions and refusals, represented by sets of refused actions, occur. To express this, the transition relation $\longrightarrow$ is extended with refusal transitions: self-loop transitions labelled with a set of actions $A \subseteq L$, expressing that all actions in A can be refused. The transition relation $\Longrightarrow$ (definition 3.2.3) is then extended analogously to $\stackrel{\varphi}{\Longrightarrow}$ with $\varphi \in (L \cup \mathcal{P}(L))^*$.

Definition 4.3.6
Let $p \in \mathcal{LTS}(L)$ and $A \subseteq L$.

1. $p \xrightarrow{A} p' \ =_{\text{def}} \ p = p'$ and $\forall \mu \in A \cup \{\tau\} :\ p \xrightarrow{\mu}\!\!\!\!\!/\ $
2. The *failure traces* of p are: $Ftraces(p) \ =_{\text{def}} \ \{\ \varphi \in (L \cup \mathcal{P}(L))^* \mid p \stackrel{\varphi}{\Longrightarrow}\}$

□

Proposition 4.3.7
$i \leq_{rf} s$ iff $Ftraces(i) \subseteq Ftraces(s)$

□

We conclude this section by relating the implementation relations based on symmetric interactions and illustrating them using the candy machines in Figure 4.3. These examples also illustrate the inequalities of proposition 4.3.8.

Proposition 4.3.8

1. $\leq_{tr},\ \leq_{te},\ \leq_{rf}$ are preorders; **conf** is reflexive, but not transitive.
2. $\leq_{rf} \ \subset \ \leq_{te} \ = \ \leq_{tr} \cap \textbf{conf}$

□

Example 4.3.9
Figure 4.3 illustrates the implementation relations discussed in this section. □

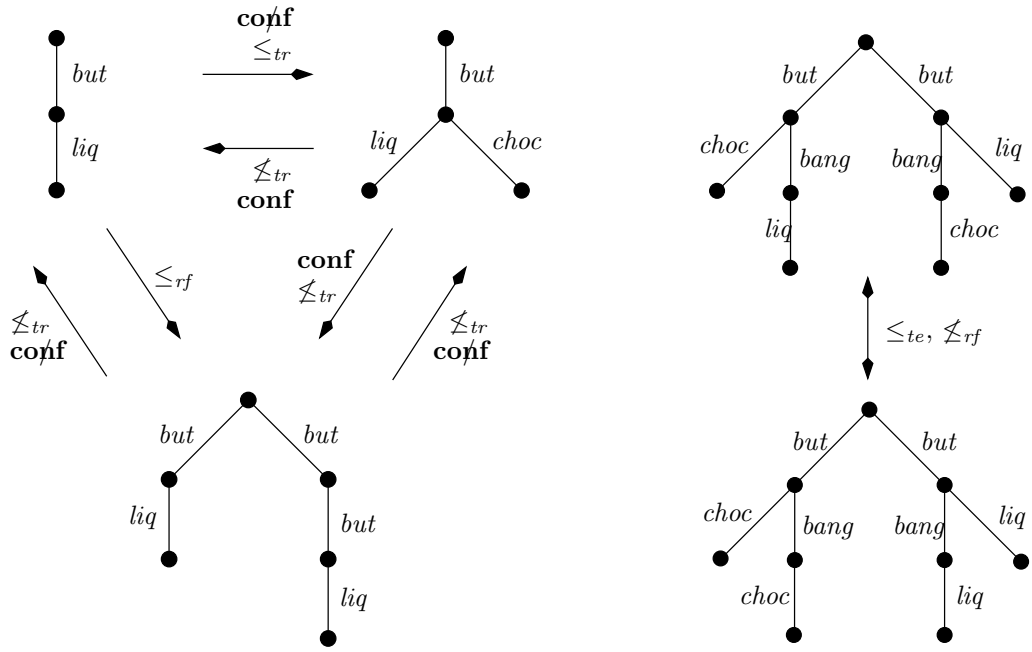


Figure 4.3:

4.4 Relations based on Inputs and Outputs

We now make the assumption that implementations can be modelled as input-output transition systems. Specifications, being more abstract than implementations, are not necessarily written in such a way that input actions can never be refused. We still allow specifications to be labelled transition systems: we consider implementation relations **imp** $\subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$. Like the relations based on symmetric interactions in Section 4.3, we define them extensionally following (4.1).

4.4.1 The Input-Output Testing Relation

The implementation relations $\leq_{te}$ and **conf** were defined by relating the observations made of the implementation by a symmetrically interacting observer $u \in \mathcal{LTS}(L)$ to the observations made of the specification (definitions 4.3.1 and 4.3.3). An analogous testing scenario can be defined for input-output transition systems using the fact that communication takes place along the lines explained in Section 3.4: the input actions of the observer synchronize with the output actions of the implementation, and vice versa, so an input-output implementation in $\mathcal{IOTS}(L_I, L_U)$ communicates with an 'output-input' observer in $\mathcal{IOTS}(L_U, L_I)$, see Figure 4.4. In this way the *input-output testing relation* $\leq_{iot}$ is defined between $i \in \mathcal{IOTS}(L_I, L_U)$ and $s \in \mathcal{LTS}(L_I \cup L_U)$ by requiring that any possible observation made of i by any 'output-input' transition system is a possible observation of s by the same observer (cf. definition 4.3.1).

Definition 4.4.1

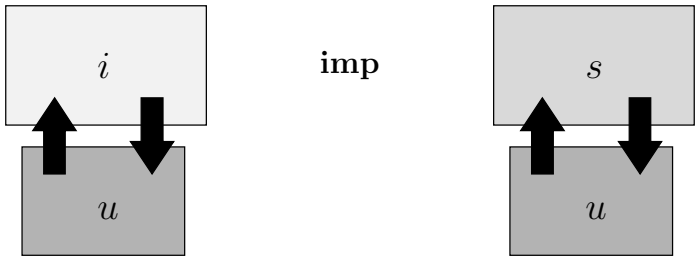


Figure 4.4: The inputs of i and s are the outputs of the environment u and vice versa

Let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$, then

$$i \leq_{iot} s \;=_{\text{def}}\; \forall u \in \mathcal{IOTS}(L_U, L_I) : obs_c(u,i) \subseteq obs_c(u,s) \text{ and } obs_t(u,i) \subseteq obs_t(u,s)$$

□

Note that, despite what was said above about the communication between the implementation and the observer, the observations made of s are based on the communication between an input-output transition system and a standard labelled transition system, since s need not be an input-output system. Technically there is no problem in making such observations: the definitions of obs_c, obs_t, $\|$, and $\,.\,\textbf{after}\,.\,\textbf{deadlocks}$ apply to labelled transition systems, not only to input-output transition systems. Below we will elaborate on this possibility to have $s \notin \mathcal{IOTS}$.

In [Seg93] the testing scenario of testing preorder [DNH84, DN87] was applied to define a relation on Input/Output Automata completely analogous to definition 4.4.1. It was shown to yield the implementation relation *quiescent trace preorder* introduced in [Vaa91]. Although we are more liberal with respect to the specification, $s \in \mathcal{LTS}(L_I \cup L_U)$, and input-output transition systems differ marginally from Input/Output Automata, exactly the same intensional characterization is obtained: $\leq_{iot}$ is fully characterized by trace inclusion and inclusion of quiescent traces.

Proposition 4.4.2
$i \leq_{iot} s$ iff $traces(i) \subseteq traces(s)$ and $Qtraces(i) \subseteq Qtraces(s)$ □

Sketch of the proof
Comparing with the analogous definition and proposition for $\leq_{te}$ (definition 4.3.1 and proposition 4.3.2), we see that the observation of deadlock of $u \,\|\, i$ can only occur if i is in a state where it cannot produce any output (a quiescent state) and u is in a state where it cannot produce any input (input with respect to i). It follows then that for inclusion of obs_c it suffices to consider only quiescent traces. Inclusion of obs_t corresponds to inclusion of traces. □

Comparing the intensional characterization of $\leq_{iot}$ in proposition 4.4.2 with the one for $\leq_{te}$ (proposition 4.3.2), we see that the restriction to input-output systems simplifies the corresponding intensional characterization. Instead of sets of pairs consisting of a trace and a set of actions (failure pairs), it suffices to look at just two sets of traces.

Another characterization of $\leq_{iot}$ can be given by collecting for each trace all possible outputs that a process may produce after that trace, including the special action δ that indicates possible quiescence. This is the set *out*, cf. Definition 3.4.5 in Section 3.4. Proposition 4.4.3 then states that an implementation is correct according to $\leq_{iot}$ if all outputs it can produce after any trace σ can also be produced by the specification. Since this also holds for δ, the implementation may show no outputs only if the specification can do so.

Proposition 4.4.3
$i \leq_{iot} s$ iff $\forall \sigma \in L^* : \ out(\, i \,\mathbf{after}\, \sigma\,) \ \subseteq \ out(\, s \,\mathbf{after}\, \sigma\,)$

□

Sketch of the proof
Using the facts that $\sigma \in Qtraces(p)$ iff $\delta \in out(\, p \,\mathbf{after}\, \sigma\,)$ and $\sigma \in traces(p)$ iff $out(\, p \,\mathbf{after}\, \sigma\,) \neq \emptyset$, the proposition follows immediately from proposition 4.4.2.

□

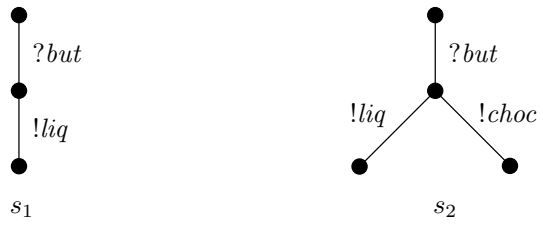


Figure 4.5: Two non-input-output specifications

Example 4.4.4
Using proposition 4.4.3, it follows that $q_1 \leq_{iot} q_2$ (Figure 3.2): an implementation capable of only producing *liquorice* conforms to a specification that prescribes to produce either *liquorice* or *chocolate*. Although q_2 looks deterministic, in fact it specifies that after *button* there is a nondeterministic choice between supplying *liquorice* or *chocolate*. This also implies that q_2 is equivalent to $?but;!liq;\mathbf{stop} \ \Box \ ?but;!choc;\mathbf{stop}$ (omitting the input self-loops) for this kind of testing. Such an equivalence does not hold for $\leq_{te}$ in the symmetric case. If we want to specify a machine that produces both *liquorice* and *chocolate*, then two buttons are needed to select the respective candies:

$$?\textit{liq-button} \ ; \ !liq \ ; \ \mathbf{stop} \ \ \Box \ \ ?\textit{choc-button} \ ; \ !choc \ ; \ \mathbf{stop}$$

On the other hand, $q_2 \not\leq_{iot} q_1, q_3$: if the specification prescribes to produce only *liquorice*, then an implementation should not have the possibility to produce *chocolate*: $!choc \in out(\, q_2 \,\mathbf{after}\, ?but\,)$, while $!choc \notin out(\, q_1 \,\mathbf{after}\, ?but\,)$ and also $!choc \notin out(\, q_3 \,\mathbf{after}\, ?but\,)$.

We have $q_1 \leq_{iot} q_3$, but $q_3 \not\leq_{iot} q_1, q_2$, since q_3 may refuse to produce anything after the *button* has been pushed once, while both q_1 and q_2 will always output something. Formally: $\delta \in out(\, q_3 \,\mathbf{after}\, ?but\,)$, while $\delta \notin out(\, q_1 \,\mathbf{after}\, ?but\,), out(\, q_2 \,\mathbf{after}\, ?but\,)$.

Figure 4.5 presents two non-input-output transition system specifications, but none of q_1, q_2, q_3 correctly implements s_1 or s_2; the problem occurs with non-specified input traces of the specification:
$out(\, q_1 \,\mathbf{after}\, ?but{\cdot}?but\,), \ out(\, q_2 \,\mathbf{after}\, ?but{\cdot}?but\,), \ out(\, q_3 \,\mathbf{after}\, ?but{\cdot}?but\,) \neq \emptyset,$

while $?but{\cdot}?but \notin traces(s_1)$, $traces(s_2)$,
so $out(\, s_1 \textbf{ after } ?but{\cdot}?but\,)$, $out(\, s_2 \textbf{ after } ?but{\cdot}?but\,) = \emptyset$.

□

4.4.2 The *ioconf* Relation

The relation $\leq_{iot}$ does not require the specification to be an input-output transition system: a specification may have states that can refuse input actions. Such a specification is interpreted as an incompletely specified input-output transition system, i.e., a transition system where a distinction is made between inputs and outputs, but where some inputs are not specified in some states. The intention of such specifications often is that the specifier does not care about the responses of an implementation to such non-specified inputs. If a candy machine is specified to deliver liquorice after a button is pushed as in s_1 in Figure 4.5, then it is intentionally left open what an implementation may do after the button is pushed twice: perhaps ignoring it, supplying one of the candies, or responding with an error message. Intuitively, q_1 would conform to s_1; however, $q_1 \not\leq_{iot} s_1$, as was shown in example 4.4.4. The implementation freedom, intended with non-specified inputs, cannot be expressed with the relation $\leq_{iot}$. From proposition 4.4.3 the reason can be deduced: since any implementation can always perform any sequence of input actions, and since from definition 3.4.5 it is easily deduced that $out(\, p \textbf{ after } \sigma\,) \neq \emptyset$ iff $p \stackrel{\sigma}{\Longrightarrow}$, we have that an $\leq_{iot}$-implementable specification must at least be able to perform any sequence of input actions. So the specification must be an input-output transition system, too, otherwise no $\leq_{iot}$-implementation can exist.

For Input/Output Automata a solution to this problem is given in [DNS95], using the so-called demonic semantics for process expressions. In this semantics a transition to a demonic process Ω is added for each non-specified input. Since Ω exhibits any behaviour, the behaviour of the implementation is not prescribed after such a non-specified input. We choose another solution to allow for non-input-output transition system specifications to express implementation freedom for non-enabled inputs: we introduce a weaker implementation relation. The discussion above immediately suggests how such a relation can be defined: instead of requiring inclusion of *out*-sets for all traces in L^* (proposition 4.4.3), the weaker relation requires only inclusion of *out*-sets for traces that are explicitly specified in the specification. This relation is called *i/o-conformance* **ioconf**, and, analogously to **conf**, it allows partial specifications which only state requirements for traces explicitly specified in the specification (cf. the relation between $\leq_{te}$ and **conf**, definitions 4.3.1 and 4.3.3, and propositions 4.3.2 and 4.3.4).

Definition 4.4.5
Let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$, then

$$i \textbf{ ioconf } s \quad =_{\text{def}} \quad \forall \sigma \in traces(s) : \quad out(\, i \textbf{ after } \sigma\,) \;\subseteq\; out(\, s \textbf{ after } \sigma\,)$$

□

Example 4.4.6
Consider again Figures 3.2 and 4.5. Indeed, we have q_1 **ioconf** s_1, whereas we had $q_1 \not\leq_{iot} s_1$. According to **ioconf**, s_1 specifies only that after one *button*, *liquorice* must be produced; with **ioconf**, s_1 does not care what happens if the *button* is pushed twice, as was the case with $\leq_{iot}$.

On the other hand, q_2 **ioco⁄nf** s_1, since q_2 can produce more than *liquorice* after the *button* has been pushed: $out(\, q_2 \textbf{ after } ?but\,) = \{!liq, !choc\} \not\subseteq \{!liq\} = out(\, s_1 \textbf{ after } ?but\,)$. Moreover, q_1, q_2 **ioconf** s_2, but q_3 **ioco⁄nf** s_1, s_2, since $\delta \in out(\, q_3 \textbf{ after } ?but\,)$, while $\delta \notin out(\, s_1 \textbf{ after } ?but\,)$, $out(\, s_2 \textbf{ after } ?but\,)$.

□

4.4.3 The Input-Output Refusal Relation

We have seen implementation relations with symmetric interactions based on observers without and with θ-label, which resulted in the relations $\leq_{te}$ and $\leq_{rf}$, respectively, and we have seen an implementation relation with inputs and outputs based on observers without θ-label. Naturally, the next step is an implementation relation with inputs and outputs based on observers with the power of the θ-label. The resulting relation is called the *input-output refusal relation* $\leq_{ior}$.

Definition 4.4.7
Let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$, then

$$i \leq_{ior} s \quad =_{\text{def}} \quad \forall u \in \mathcal{IOTS}(L_U, L_I \cup \{\theta\}) : \quad \begin{array}{l} obs_c^{\theta}(u,i) \subseteq obs_c^{\theta}(u,s) \text{ and} \\ obs_t^{\theta}(u,i) \subseteq obs_t^{\theta}(u,s) \end{array}$$

□

A quiescent trace was introduced as a trace ending in the absence of outputs. Taking the special action δ a quiescent trace $\sigma \in L^*$ can be written as a δ-ending trace $\sigma{\cdot}\delta \in (L \cup \{\delta\})^*$. Here, the special action δ appears always as the last action in the trace.

If this special action δ is treated as a completely normal action which can occur at any place in a trace, we obtain suspension traces, cf. Definition 3.4.4 in Section 3.4. In suspension traces quiescence can occur repetitively, hence the alternative name *repetitive quiescent trace*.

The implementation relation $\leq_{ior}$ turns out to be characterized by inclusion of suspension traces (and hence it could also be called *repetitive quiescence relation*). Since quiescence corresponds to a refusal of L_U (definition 4.3.6), it follows that suspension traces are exactly the failure traces in which only L_U occurs as refusal set, i.e., failure traces restricted to $(L \cup \{L_U\})^*$, and where δ is written for the refusal L_U.

Proposition 4.4.8

1. $Straces(p) \;=\; Ftraces(p) \;\cap\; (L \cup \{L_U\})^*$, where L_U is written as δ.
2. $i \leq_{ior} s$ iff $Straces(i) \subseteq Straces(s)$

□

Sketch of the proof
For part 2, analogous to the proof of proposition 4.4.2, and comparing with the corresponding situation for $\leq_{rf}$ (definition 4.3.5 and proposition 4.3.7), a refusal can only be observed if i is in a state where it cannot produce any output (a quiescent state) and u is in a state where it cannot produce any input (input with respect to i). So the only refusals of i that can be observed are L_U. As opposed to proposition 4.4.2, the normal traces are included in the suspension traces, so they need not be mentioned separately in the proposition.

□

An intensional characterization of $\leq_{ior}$ in terms of *out*-sets, analogous to proposition 4.4.3, is easily given by generalizing the definition of **after** (definition 3.2.5) in a straightforward way to suspension traces.

Proposition 4.4.9
$i \leq_{ior} s$ iff $\forall \sigma \in L_\delta^* : \; out(\, i \,\mathbf{after}\, \sigma\,) \subseteq out(\, s \,\mathbf{after}\, \sigma\,)$

□

4.5 The *ioco* Relation

A basic relation for testing and algorithmic test derivation is **ioco**. The relation **ioco**, abbreviated from *i*nput-*o*utput *co*nformance, defines when an implementation is correct with respect to, or conforming to its specification. For **ioco** it is assumed that the specification is a labelled transition system where inputs and outputs can be distinguished: $s \in \mathcal{LTS}(L_I \cup L_U)$. The implementation is assumed to be modelled as an input-output transition system: $i \in \mathcal{IOTS}(L_I, L_U)$; so, $\mathbf{ioco} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$.

An implementation i is **ioco**-correct with respect to a specification s if any 'output' produced by the implementation after a suspension trace σ can be produced by the specification after the same trace σ. Any 'output' includes the absence of outputs as modelled by quiescence δ, cf. Definition 3.4.3.

Definition 4.5.1
$i \;\mathbf{ioco}\; s \quad =_{\text{def}} \quad \forall \sigma \in Straces(s) : \; out(\, i \,\mathbf{after}\, \sigma\,) \subseteq out(\, s \,\mathbf{after}\, \sigma\,)$

□

The relation **ioco** is an important implementation relation. On the one hand it has a sound theoretical background, while on the other hand it is an intuitive notion of what a correct implementation should do and should not do. The relation **ioco** will be the basis for testing and test generation in Chapter 5.

Example 4.5.2
Consider Figure 3.2 and Example 3.4.6. Using Definition 4.5.1, we have that q_1 **ioco** q_2: an implementation capable of only producing *liquorice* conforms to a specification that prescribes to produce either *liquorice* or *chocolate*. Although q_2 looks deterministic, in fact it specifies that after *button* there is a nondeterministic choice between supplying *liquorice* or *chocolate*.

If we want to specify a machine that produces both *liquorice* and *chocolate*, then two buttons are needed to select the respective candies:

$$?\mathit{liq\text{-}button} \;;\; !\mathit{liq} \;;\; \mathbf{stop} \;\Box\; ?\mathit{choc\text{-}button} \;;\; !\mathit{choc} \;;\; \mathbf{stop}$$

On the other hand, q_2 **io$\not$o** q_1 and q_2 **io$\not$o** q_3: if the specification prescribes to produce only *liquorice* then an implementation shall not have the possibility to produce *chocolate*: for the implementation $!\mathit{choc} \in out(\, q_2 \,\mathbf{after}\, ?\mathit{but}\,)$, while for the specifications: $!\mathit{choc} \notin out(\, q_1 \,\mathbf{after}\, ?\mathit{but}\,)$ and $!\mathit{choc} \notin out(\, q_3 \,\mathbf{after}\, ?\mathit{but}\,)$.

We have q_1 **ioco** q_3, but q_3 **io$\not$o** q_1 and q_3 **io$\not$o** q_2, since q_3 may refuse to produce anything after the *button* has been pushed once, while both q_1 and q_2 will always output something. Formally: $\delta \in out(\, q_3 \,\mathbf{after}\, ?\mathit{but}\,)$, while $\delta \notin out(\, q_1 \,\mathbf{after}\, ?\mathit{but}\,)$ and $\delta \notin out(\, q_2 \,\mathbf{after}\, ?\mathit{but}\,)$.

Figure 4.5 presents two non-input-output transition system specifications. We have q_1 **ioco** s_1; s_1 does not care what happens after the *button* has been pushed twice, since for the specification $?but{\cdot}?but \notin Straces(s_1)$.

On the other hand, q_2 **io$\not$o** s_1, since q_2 can produce more than *liquorice* after the *button* has been pushed: $out(\, q_2 \textbf{ after } ?but\,) = \{!liq, !choc\} \not\subseteq \{!liq\} = out(\, s_1 \textbf{ after } ?but\,)$.

Moreover, q_1 **ioco** s_2 and q_2 **ioco** s_2, but q_3 **io$\not$o** s_1 and q_3 **io$\not$o** s_2, since $\delta \in out(\, q_3 \textbf{ after } ?but\,)$, while $\delta \notin out(\, s_1 \textbf{ after } ?but\,)$ and $\delta \notin out(\, s_2 \textbf{ after } ?but\,)$. □

When **ioco** is compared with the relations introduced in Sections 4.3 and 4.4 it can be observed that its definition stems from Proposition 4.4.9 by restricting inclusion of *out*-sets to suspension traces of the specification, using completely analogous arguments as in the definitions of **conf** (Definition 4.3.3) and **ioconf** (Definition 4.4.5). It only requires an implementation to react correctly to the traces that are explicitly specified in the specification, and it leaves freedom to an implementation to react in any manner to traces not explicitly specified.

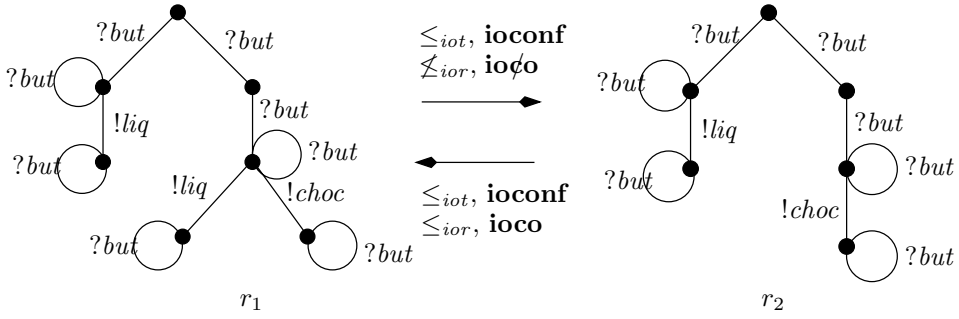


Figure 4.6: The difference between $\leq_{iot}$ and $\leq_{ior}$

Example 4.5.3
The difference between $\leq_{iot}$ and $\leq_{ior}$, and between **ioconf** and **ioco** is illustrated with the processes r_1 and r_2 in Figure 4.6: r_1 **ioconf** r_2, but r_1 **io$\not$o** r_2;
$out(\, r_1 \textbf{ after } ?but{\cdot}\delta{\cdot}?but\,) = \{!liq, !choc\}$ and $out(\, r_2 \textbf{ after } ?but{\cdot}\delta{\cdot}?but\,) = \{!choc\}$. Intuitively, after pushing the *button*, we observe that nothing is produced by the machine, so we push the *button* again. Machine r_1 may then produce either *liquorice* or *chocolate*, while machine r_2 will always produce *chocolate*. When we use the relation **ioconf**, the observation always terminates after observing that nothing is produced. Hence, there is no way to distinguish between entering the left or the right branch of r_1 or r_2; after the *button* is pushed twice, both machines may produce either *liquorice* or *chocolate*: $out(\, r_{1,2} \textbf{ after } ?but{\cdot}?but\,) = \{!liq, !choc\}$. □

4.6 Relating Relations with Inputs and Outputs

Two kinds of observations were used in the extensional definitions of testing preorder $\leq_{te}$ (definition 4.3.1), refusal preorder $\leq_{rf}$ (definition 4.3.5), the input-output test-

ing relation $\leq_{iot}$ (definition 4.4.1), and the input-output refusal relation $\leq_{ior}$ (definition 4.4.7): the traces and the completed traces of the composition of a process and an observer, expressed by $obs_t(u,p)$ and $obs_c(u,p)$, respectively. The varying parameters in defining these four relations were the distinction between inputs and outputs (and associated input-enabledness) and the ability to observe refusals by adding the θ-label to the class of observers.

	$u \in \mathcal{LTS}(L)$	$u \in \mathcal{LTS}(L_\theta)$
no inputs and no outputs	$\leq_{te}$ obs_c	$\leq_{rf}$ obs_c^θ or obs_t^θ
inputs and outputs	$\leq_{iot}$ obs_c and obs_t	$\leq_{ior}$ obs_t^θ

Table 4.1: Observations obs_c and obs_t

Although all four relations were defined by requiring inclusion of both obs_c and of obs_t, some of the relations only need observations of one kind. This is indicated in table 4.1 by mentioning the necessary and sufficient observations. Adding the ability to observe refusals to observers, using the θ-action, makes observation of completed traces with obs_c superfluous: for $\leq_{rf}$ and $\leq_{ior}$ it suffices to consider observations of the kind obs_t. If no distinction between inputs and outputs is made, any observation of a trace in obs_t can always be simulated in obs_c with an observer which can perform only this particular trace and then terminates: for $\leq_{te}$ and $\leq_{rf}$ it suffices to consider observations of the kind obs_c. Only for $\leq_{iot}$ both kinds of observations are necessary, as shows the example in Figure 4.7. Let $L_I = \emptyset$ and $L_U = \{x, y\}$; then, to define both intuitively incorrect implementations i_1 and i_2 as not $\leq_{iot}$-related, we need both obs_c and obs_t: $\forall u \in \mathcal{IOTS}(L_U, L_I) : obs_t(u, i_1) \subseteq obs_t(u, s)$, while $\forall u \in \mathcal{IOTS}(L_U, L_I) : obs_c(u, i_2) \subseteq obs_c(u, s)$.

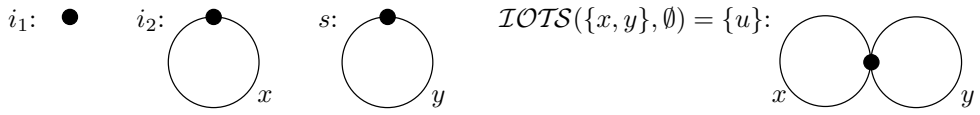


Figure 4.7: Observations for $\leq_{iot}$

The input-output implementation relations defined so far, viz. $\leq_{iot}$, **ioconf**, $\leq_{ior}$ and **ioco**, are easily related using their characterizations in terms of *out*-sets. The only difference between the relations is the set of (suspension) traces for which the *out*-sets are compared (cf. Propositions 4.4.3 ($\leq_{iot}$) and 4.4.9 ($\leq_{ior}$), and definitions 4.4.5 (**ioconf**) and 4.5.1 (**ioco**)). So if we introduce the following class of relations $\mathbf{ioco}_\mathcal{F}$ with $\mathcal{F} \subseteq L_\delta^*$:

$$i \ \mathbf{ioco}_\mathcal{F} \ s \quad =_{\text{def}} \quad \forall \sigma \in \mathcal{F} : \ out(\, i \ \mathbf{after} \ \sigma \,) \ \subseteq \ out(\, s \ \mathbf{after} \ \sigma \,) \tag{4.3}$$

then they can all be expressed as instances of $\mathbf{ioco}_{\mathcal{F}}$:

$$\begin{array}{lcllcl} \leq_{iot} & = & \mathbf{ioco}_{L^*} & \mathbf{ioconf} & = & \mathbf{ioco}_{traces(s)} \\ \leq_{ior} & = & \mathbf{ioco}_{L^*_\delta} & \mathbf{ioco} & = & \mathbf{ioco}_{Straces(s)} \end{array} \tag{4.4}$$

Using (4.3) and (4.4) the input-output implementation relations are easily related by relating their respective sets $\mathcal{F}$ (Proposition 4.6.1). The inequalities follow from the candy machines in Examples 4.4.6, 4.5.2 and 4.5.3.

Proposition 4.6.1

$$\leq_{ior} \subset \left\{ \begin{array}{l} \leq_{iot} \\ \mathbf{ioco} \end{array} \right\} \subset \mathbf{ioconf}$$

□

4.7 Correctness in the Testing Framework

Implementation relation The implementation relation **imp** in the formal testing framework of Chapter 2 is instantiated with the relation $\mathbf{ioco} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$ (definition 4.5.1). This means that the notion of correctness for which tests will be derived is formalized as **ioco**. This relation expresses intuitively what a correct implementation i of a specification s is:

- i may only produce outputs which are allowed by s, i.e., any output produced by i must also be producible by s in the same situation, i.e., after the same suspension trace; and
- if s specifies that some reaction must be produced (i.e., s is not *quiescent*; $\delta \notin out(\, s \,\mathbf{after}\, \sigma\,)$), then i must produce some output, i.e., i may not be quiescent.

Chapter 5

Testing Transition Systems

5.1 Introduction

Now that we have formal specifications – expressed as labelled transition systems –, implementations – modelled by input-output transition systems – and a formal definition of conformance – expressed by the implementation relation **ioco** – we can start the discussion of conformance testing. The first point of discussion is what these test cases look like, how they are executed, and when they are successful. Then an algorithm for test derivation is presented.

5.2 Testing Input-Output Transition Systems

Tests A test case is a specification of the behaviour of a tester in an experiment to be carried out with an implementation under test. Such behaviour, like other behaviours, can be described by a labelled transition system. When testing for **ioco** we should be able to observe quiescence. For this, we add a special label θ $(\theta \notin L \cup \{\tau, \delta\})$. The occurrence of θ in a test models the detection of quiescence in an implementation, i.e., the observation that no output is produced. So, tests will be processes in $\mathcal{LTS}(L_I \cup L_U \cup \{\theta\})$.

But there is more to require from a transition system representing a test. To guarantee that the experiment lasts for a finite time, a test case should have finite behaviour. Moreover, a tester executing a test case would like to have control over the testing process as much as possible, so a test case should be specified in such a way that unnecessary nondeterminism is avoided. First of all, this implies that the test case itself must be deterministic. But also we will not allow test cases with a choice between an input action and an output action, nor a choice between multiple input actions (input and output from the perspective of the implementation). Both introduce unnecessary nondeterminism in the test run: if a test case can offer multiple input actions, or a choice between input and output, then the continuation of the test run is unnecessarily nondeterministic, since any input-output implementation can always accept any input action. This implies that a state of a test case either is a terminal state, or offers one particular input to the implementation, or accepts all possible outputs from the

implementation, including the δ-action which is accepted by a θ-action in the test case. Finally, to be able to decide about the success of a test, the terminal states of a test case are labelled with the verdict **pass** or **fail**. Altogether, we come to the following definition of the class of test cases $\mathcal{TEST}$. This class instantiates the set $TESTS$ of test cases in the generic testing framework.

Definition 5.2.1

1. A *test case* t is a labelled transition system $\langle S, L_I \cup L_U \cup \{\theta\}, T, s_0 \rangle$ such that
   - t is deterministic and has finite behaviour;
   - S contains terminal states **pass** and **fail**, with $init(\mathbf{pass}) = init(\mathbf{fail}) = \emptyset$;
   - for any state $t' \in S$ of the test case, $t' \neq \mathbf{pass}, \mathbf{fail}$, either $init(t') = \{a\}$ for some $a \in L_I$, or $init(t') = L_U \cup \{\theta\}$.

   The class of test cases over L_U and L_I is denoted as $\mathcal{TEST}(L_U, L_I)$.
2. A *test suite* T is a set of test cases: $T \subseteq \mathcal{TEST}(L_U, L_I)$.

□

Note that L_I and L_U refer to the inputs and outputs from the point of view of the implementation under test, so L_I denotes the outputs, and L_U denotes the inputs of the test case.

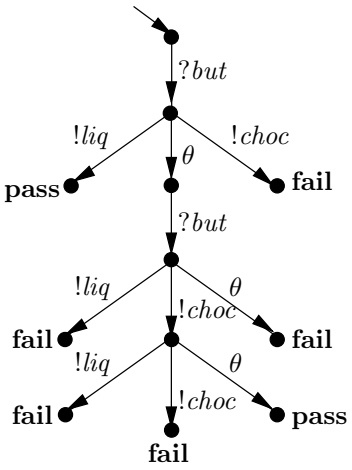


Figure 5.1: A test case t

Example 5.2.2
Figure 5.1 gives an example of a test case, which specifies that as the first action the input ?*but* must be supplied to the implementation under test. The special transitions labelled θ model the observation of quiescence, which is usually implemented as a time-out: this transition will be taken if none of the output responses can be observed; the implementation under test is quiescent.

In a TTCN-like notation (TTCN is the test notation of ISO [ISO91, part 3]), this test case could be written as in table 5.1. Note that in TTCN "!" and "?" are exchanged, i.e., "!" denotes outputs of the test case and "?" denotes inputs.

□

Test Case t – Dynamic Behaviour		
behaviour	constraints	verdict
! *but* ; start θ_1		
? *liq* ;		**pass**
? *choc* ;		**fail**
? time out θ_1		
! *but* ; start θ_2		
? *liq* ;		**fail**
? time out θ_2		**fail**
? *choc* ; start θ_3		
? *liq*		**fail**
? *choc*		**fail**
? time out θ_3		**pass**

Table 5.1: TTCN for the test case of Figure 5.1

Test execution A test run of an implementation with a test case is modelled by the synchronous parallel execution $\rceil\!|\!|$ of the test case with the implementation under test. The parallel operator $\rceil\!|\!|$ is the usual parallel operator $\|$ (Section 3.3) with an addition to cope with the synchronization of the implementation's δ-quiescence with the test case's θ-action. A test run continues until no more interactions are possible, i.e., until a deadlock occurs. Since for each state $t' \neq \mathbf{pass}, \mathbf{fail}$ of a test case either $\mathit{init}(t') = \{a\}$ for some $a \in L_I$, in which case no deadlock can occur because of input-enabledness of implementations, or $\mathit{init}(t') = L_U \cup \{\theta\}$, in which case no deadlock can occur because of the θ-transition, it follows that deadlock can only occur in the terminal states **pass** and **fail**. An implementation passes a test run if and only if the test run deadlocks in **pass**. Since an implementation can behave nondeterministically, different test runs of the same test case with the same implementation may lead to different terminal states and hence to different verdicts. An implementation passes a test case if and only if all possible test runs lead to the verdict **pass**. This means that each test case must be executed several times in order to give a final verdict, theoretically even infinitely many times.

Definition 5.2.3
Let $t \in \mathcal{TEST}(L_U, L_I)$ and $i \in \mathcal{IOTS}(L_I, L_U)$.

1. *Running* a test case t with an implementation i is modelled by the parallel operator $\rceil\!|\!| : \mathcal{TEST}(L_U, L_I) \times \mathcal{IOTS}(L_I, L_U) \to \mathcal{LTS}(L_I \cup L_U \cup \{\theta\})$ which is defined by the following inference rules:

$$\begin{array}{lll} i \xrightarrow{\tau} i' & \vdash & t\rceil\!|\!| i \xrightarrow{\tau} t\rceil\!|\!| i' \\ t \xrightarrow{a} t',\ i \xrightarrow{a} i',\ a \in L_I \cup L_U & \vdash & t\rceil\!|\!| i \xrightarrow{a} t'\rceil\!|\!| i' \\ t \xrightarrow{\theta} t',\ \forall \mu \in L_U \cup \{\tau\} :\ i \xrightarrow{\mu}\!\!\!\!\!/\ \ \ & \vdash & t\rceil\!|\!| i \xrightarrow{\theta} t'\rceil\!|\!| i \end{array}$$

2. A *test run* of t with i is a trace of $t\rceil\!|\!| i$ leading to a terminal state of t:

$$\sigma \text{ is a test run of } t \text{ and } i \ \ =_{\text{def}} \ \ \exists i' :\ t\rceil\!|\!| i \stackrel{\sigma}{\Longrightarrow} \mathbf{pass}\rceil\!|\!| i' \ \text{ or } \ t\rceil\!|\!| i \stackrel{\sigma}{\Longrightarrow} \mathbf{fail}\rceil\!|\!| i'$$

3. Implementation i *passes* test case t if all their test runs lead to the **pass**-state of t:

$$i \ \mathbf{passes}\ t \ \ =_{\text{def}} \ \ \forall \sigma \in L_\theta^*,\ \forall i' :\ t\rceil\!|\!| i \stackrel{\sigma}{\not\Longrightarrow} \mathbf{fail}\rceil\!|\!| i'$$

4. An implementation i *passes* a test suite T if it passes all test cases in T:

$$i \textbf{ passes } T \quad =_{\text{def}} \quad \forall t \in T : \; i \textbf{ passes } t$$

If i does not pass the test suite, it fails: $i \textbf{ fails } T \;\; =_{\text{def}} \;\; \exists t \in T : i \textbf{ passes } t$. □

Observations When defining test runs, test execution and passing a test suite explicitly in terms of the formal testing framework, we start with observations *OBS*. Observations can be defined as logs of actions, i.e., traces over $L \cup \{\theta\}$, so *OBS* is instantiated with $(L \cup \{\theta\})^*$.

Observation function The observation function *obs* is instantiated with the collection of test runs that can be obtained for particular t and i:

$$obs(t, i) \quad =_{\text{def}} \quad \{ \; \sigma \in (L \cup \{\theta\})^* \;\; | \;\; \sigma \text{ is a test run of } t \text{ and } i \; \}$$

Example 5.2.4
For r_1 of Figure 4.6 there are three observations with t of Figure 5.1:

$$t \rceil\!\rceil \, r_1 \xRightarrow{?but \cdot !liq} \textbf{pass} \lceil\!\lceil \, r_1'$$
$$t \rceil\!\rceil \, r_1 \xRightarrow{?but \cdot \theta \cdot ?but \cdot !liq} \textbf{fail} \lceil\!\lceil \, r_1''$$
$$t \rceil\!\rceil \, r_1 \xRightarrow{?but \cdot \theta \cdot ?but \cdot !choc \cdot \theta} \textbf{pass} \lceil\!\lceil \, r_1'''$$

where r_1', r_1'', and r_1''' are the leaves of r_1 from left to right.
So, $obs(t, r_1) = \{ \; ?but \cdot !liq, \; ?but \cdot \theta \cdot ?but \cdot !liq, \; ?but \cdot \theta \cdot ?but \cdot !choc \cdot \theta \; \}$.
□

Verdicts An implementation i passes a test case t if all their test runs lead to the **pass**-state of t (Definition 5.2.3.2). This can be rephrased in terms of the testing framework by defining the verdict assigned to a set of observations $O \subseteq OBS$ as:

$$\nu_t(O) \quad =_{\text{def}} \quad \begin{cases} \textbf{pass} & \text{if } \forall \sigma \in O : t \xRightarrow{\sigma} \textbf{pass} \\ \textbf{fail} & \text{otherwise} \end{cases}$$

Example 5.2.5
Continuing example 5.2.4 we have that, since the terminal state of t for the second run is **fail**, the verdict for r_1 is **fail**: r_1 **fails** t. Similarly, it can be checked that the verdict for r_2 is **pass**.
□

5.3 Test Generation for Input-Output Transition Systems

Now all ingredients are there to present an algorithm to generate test suites from labelled transition system specifications for the implementation relation **ioco**. A generated test suite must test implementations for conformance with respect to s and **ioco**. Ideally, an implementation should pass the test suite if and only if it conforms, but in practice this is usually not feasible. So we have to restrict to sound test suites that can only detect non-conformance, but that cannot assure conformance, see Section 2.2.

Definition 5.3.1
Let s be a specification and T a test suite; then for implementation relation **ioco**:

$$\begin{array}{llllll} T \text{ is complete} & =_{\text{def}} & \forall i: & i \textbf{ ioco } s & \text{iff} & i \textbf{ passes } T \\ T \text{ is sound} & =_{\text{def}} & \forall i: & i \textbf{ ioco } s & \text{implies} & i \textbf{ passes } T \\ T \text{ is exhaustive} & =_{\text{def}} & \forall i: & i \textbf{ ioco } s & \text{if} & i \textbf{ passes } T \end{array}$$

□

We aim at producing sound test suites from a specification s. To get some idea what such test cases will look like, we consider the definition of **ioco** (definition 4.5.1). We see that to test for **ioco** we have to check whether $out(\, i \textbf{ after } \sigma \,) \subseteq out(\, s \textbf{ after } \sigma \,)$ and this has to be done for each $\sigma \in traces(s)$. Basically, this can be done by having a test case t that executes σ:

$$t \rceil\!| \, i \stackrel{\sigma}{\Longrightarrow} t' \rceil\!| \, i'$$

and after that it should check $out(\, i \textbf{ after } \sigma \,)$ by having transitions to **pass**-states for all allowed outputs (those in $out(\, s \textbf{ after } \sigma \,)$) and transitions to **fail**-states for all erroneous outputs (those not in $out(\, s \textbf{ after } \sigma \,)$). Special care should be taken for the special output δ: δ actually models the absence of any output, so no transition will be made by the implementation if i' 'outputs' δ. This matches a θ-transition in the test case, which exactly occurs if nothing else can happen. This θ-transition will go the **pass**-state if quiescence is allowed ($\delta \in out(\, s \textbf{ after } \sigma \,)$) and to the **fail**-state if the specification does not allow quiescence at that point. All this is reflected in the following recursive algorithm for test generation for relations **ioco**. The algorithm is nondeterministic in the sense that in each recursive step it can be continued in many different ways: termination of the test case in choice 1, any input action satisfying the requirement of choice 2, or checking the allowed outputs in choice 3. Each continuation will result in another sound test case (theorem 5.3.3.1), and all possible test cases together form an exhaustive (and thus complete) test suite (theorem 5.3.3.2). The latter implies that there are no errors in an implementation that are principally undetectable with test suites generated with the algorithm. However, if the behaviour of the specification is infinite, the algorithm allows construction of infinitely many different test cases, which can be arbitrarily long, but which all have finite behaviour.

Algorithm 5.3.2
Let $s \in \mathcal{LTS}(L)$ be a specification with initial state s_0. Let S be a non-empty set of states, with initially $S = \{s_0\}$. (S represents the set of all possible states in which the implementation can be at the current stage of the test case.)

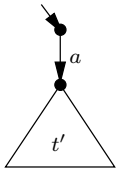
A test case $t \in \mathcal{TEST}(L_U, L_I)$ is obtained from S by a finite number of recursive applications of one of the following three nondeterministic choices:

1. 

   t := **pass**

   The single-state test case **pass** is always a sound test case. It stops the recursion in the algorithm, and thus terminates the test case.

2.

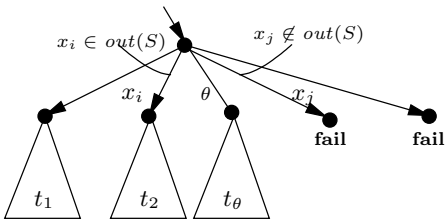


$t \;:=\; a \;;\; t'$

where $a \in L_I$, $S \,\mathbf{after}\, a \neq \emptyset$, and t' is obtained by recursively applying the algorithm for $S' = S \,\mathbf{after}\, a$.

Test case t supplies the input a to the implementation and subsequently behaves as test case t'. t' is obtained by applying the algorithm recursively to S' which is the set of specification states which can be reached via an a transition from some state in S.

3.



$$
\begin{array}{lll}
t & := & \Sigma \{\; x \;;\; \mathbf{fail} \mid x \in L_U,\; x \notin out(S) \;\} \\
 & \Box & \Sigma \{\; \theta \;;\; \mathbf{fail} \mid \delta \notin out(S) \;\} \\
 & \Box & \Sigma \{\; x \;;\; t_x \mid x \in L_U,\; x \in out(S) \;\} \\
 & \Box & \Sigma \{\; \theta \;;\; t_\theta \mid \delta \in out(S) \;\}
\end{array}
$$

where t_x and t_θ are obtained by recursively applying the algorithm for $S \,\mathbf{after}\, x$ and $S \,\mathbf{after}\, \delta$, respectively.

Test case t checks the next output of the implementation; if it is an unvalid response, i.e., $x \notin out(S)$ then the test case terminates in **fail**; if it is a valid response the test case continues recursively. The observation of quiescence δ is treated separately by the time-out action θ.

□

Now the requirements of soundness and completeness (Section 2.2) can indeed be proved.

Theorem 5.3.3
Let $s \in \mathcal{LTS}(L_I \cup L_U)$ be a specification and T_s the set of all test cases that can be generated from s with algorithm 5.3.2; let $der : \mathcal{LTS}(L_I \cup L_U) \rightarrow \mathcal{P}(\mathcal{TEST})$ be a test derivation function satisfying $der(s) \subseteq T_s$; then:

1. $der(s)$ is sound for s with respect to **ioco**;
2. T_s is exhaustive for s with respect to **ioco**.

□

Example 5.3.4
Consider the candy machines r_1 and r_2 in Figure 4.6. We use algorithm 5.3.2 to derive

a test case from r_2. The successive choices for the recursive steps of the algorithm are:

1. First choice 2 (giving an input to the implementation) is taken: $t := ?but; t_1$
2. To obtain t_1, the next output of the implementation is checked (choice 3): $t_1 := !liq; t_{2_1} \,\square\, !choc; \mathbf{fail} \,\square\, \theta; t_{2_2}$
3. If the output was $!liq$, then we stop with the test case (choice 1): $t_{2_1} := \mathbf{pass}$
4. If no output was produced (output θ; we know that we are in the right branch of r_2), then we give a next input to the implementation (choice 2): $t_{2_2} := ?but; t_3$
5. To obtain t_3 we again check the outputs (choice 3): $t_3 := !choc; t_4 \,\square\, !liq; \mathbf{fail} \,\square\, \theta; \mathbf{fail}$
6. After the output $!choc$ we check again the outputs (choice 3) to be sure that nothing more is produced: $t_4 := !choc; \mathbf{fail} \,\square\, !liq; \mathbf{fail} \,\square\, \theta; t_5$
7. For t_5 we stop with the test case (choice 1): $t_5 := \mathbf{pass}$

After putting together all pieces, we obtain t of Figure 5.1 as a sound test case for r_2. This is consistent with the results in examples 4.5.2 and 5.2.5: r_1 **io$\not$o** r_2, r_2 **ioco** r_2, and indeed r_1 **fails** t, and r_2 **passes** t. So, test case t will detect that r_1 is not **ioco**-correct with respect to r_2.

□

5.4 Transition System Testing in the Formal Framework

In this chapter we have instantiated the remaining ingredients of the formal framework of Chapter 2: $TESTS$, OBS, ν_t, obs and $der_{\mathbf{imp}}$. Moreover, we have shown that our test derivation algorithm is sound and, when taken to the extreme, exhaustive. This concludes our instantiation of the formal testing framework with labelled transition systems.

Chapter 6

Tools and Applications

6.1 Tools

The formal test theory and the **ioco**-test derivation algorithm presented in Chapter 5 fortunately have a much wider and more practical applicability than the testing of candy machines. Different test tools have been built which implement, more or less strictly, this algorithm. These tools are able to automatically derive tests from formal system specifications. These include TVEDA [Pha94b, Cla96], TGV [FJJV96], TestComposer [KJG99], TestGen [HT99] and TORX [BFV+99].

TVEDA is a tool, developed at France Telecom CNET, which is able to generate test cases in TTCN from formal specifications in a subset of the language SDL. The formal language SDL is an ITU-T standard and is often used for specification and design of telecommunication protocols [CCI92]; TTCN is an ITU-T and ISO standard for the notation of test suites [ISO91, part 3]. France Telecom uses TVEDA to generate tests for testing of telecom products, such as ATM protocols.

Actually, it is interesting to note that the test generation algorithm of TVEDA was not based on the algorithm for **ioco**-test derivation but on the intuition and heuristics of experienced test case developers at France Telecom CNET. Only careful analysis afterwards showed that this algorithm generates test cases for an implementation relation which was called "R_1" in [Pha94b] and which turned to be almost the same as **ioco**.

The tool TGV generates tests in TTCN from LOTOS or SDL specifications. LOTOS is a specification language for distributed systems standardized by ISO [ISO89]. TGV allows test purposes to be specified by means of automata, which makes it possible to identify the parts of a specification which are interesting from a testing point of view. The prototype tools TVEDA and TGV are currently integrated into the commercial SDL tool kit *Object*GEODE as a component named TestComposer.

TestGen uses LOTOS as its input language and generates test for the verification of hardware designs and components.

Whereas TVEDA and TGV only support the test derivation process by deriving test suites and expressing them in TTCN, the tool TORX combines **ioco**-test derivation and test execution in an integrated manner. This approach, where test derivation and

test execution occur simultaneously, is called *on-the-fly testing*. Instead of deriving a complete test case, the test derivation process only derives the next test event from the specification and this test event is immediately executed. While executing a test case, only the necessary part of the test case is considered: the test case is derived *lazily* (cf. lazy evaluation of functional languages; see also [VT00]). The principle is depicted in Figure 6.1. Each time the *Tester* decides whether to trigger the IUT (Implementation Under Test) with a next stimulus or to observe the output produced by the IUT. This corresponds to the choice between 2. and 3. in the test derivation algorithm 5.3.2 of Section 5.3. If a stimulus is given to the IUT (choice 2.) the *Tester* looks into the system specification – the *specification module* – for a valid stimulus and offers this input to the IUT (after suitable translation and encoding). When the *Tester* observes an output or observes that no output is available from the IUT (i.e., quiescence (definition 3.4.3) which is usually observed as a time-out), it checks whether this response is valid according to the specification (choice 3.). This process of giving stimuli to the IUT and observing responses from the IUT can continue until an output is received which is not correct according the specification resulting in a **fail**-verdict. For a correct IUT the only limits are the capabilities of the computers on which TorX is running. Test cases of length up to 500,000 test events (stimuli and responses) have been executed completely automatically [BFV+99].

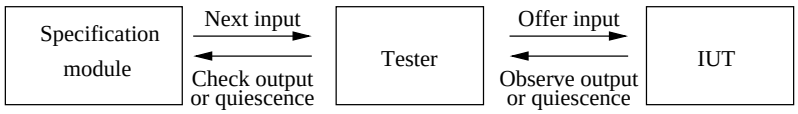


Figure 6.1: On-the-fly testing

TorX is currently able to derive test cases from Lotos and Promela specifications, i.e., the specification module in Figure 6.1 can be instantiated with a Lotos or a Promela module. The Lotos implementation is based on Cæsar [Gar98]; the Promela implementation is based on the model checker Spin [Hol91, Spi]. But since the interface between the specification module and the rest of the tool uses the Open/Cæsar interface [Gar98] for traversing through a labelled transition system, the tool can be easily extended to any formalism with transition system semantics for which there is an Open/Cæsar interface implementation available.

An important aspect in the communication between the TorX and the IUT is the encoding and decoding of test events. Test events in specifications are abstract objects which have to be translated into some concrete form of bits and bytes to communicate with the IUT and vice versa. These en-/decoding functions currently have to be written manually, still; this is a laborious task, but, fortunately, it needs to be done only once for each IUT

The tool TorX can operate in a manual or automatic mode. In the manual mode, the next test event – input or output and, if an input, the selection of the input – can be chosen interactively by the TorX user. In the automatic mode everything runs automatically and selections are made randomly. A seed for random number generation can then be supplied as a parameter to allow for repeatability of test runs. Moreover, a maximum number for the length of the test cases may be supplied. Section 6.2 will elaborate on a particular example system tested with TorX, thus illustrating the concepts presented here.

6.2 The Conference Protocol Example

One of the systems tested with the test tool TorX is a simple chat-box protocol, the *Conference Protocol* [FP95, TFPHT96]. In this case study an implementation of the conference protocol was built (in C, based on the informal description of the protocol), from which 27 mutants were derived by introducing single errors. All these 28 implementations have been tested using TorX by persons who did not know which errors had been introduced to make the mutants. This case study is described in greater detail in [BFV+99]; an elaborate description of the Conference Protocol, together with the complete formal specifications and our set of implementations can be found on the web [Pro].

This section is structured as follows. Section 6.2.1 gives an overview of the Conference Protocol and the implementations we made. Section 6.2.2 discusses the test architecture that was used for the testing activities. The testing activities themselves are described in Section 6.2.3.

6.2.1 The Conference Protocol

Informal description The conference service provides a multicast service, resembling a 'chatbox', to users participating in a conference. A conference is a group of users that can exchange messages with all conference partners in that conference. Messages are exchanged using the service primitives *datareq* and *dataind*. The partners in a conference can change dynamically because the conference service allows its users to *join* and *leave* a conference. Different conferences can exist at the same time, but each user can only participate in at most one conference at a time.

The underlying service, used by the conference protocol, is the point-to-point, connectionless and unreliable service provided by the *User Datagram Protocol* (UDP), i.e., data packets may get lost or duplicated or be delivered out of sequence but are never corrupted or misdelivered.

The object of our experiments is testing a *Conference Protocol Entity* (CPE). The CPEs send and receive *Protocol Data Units* (PDUs) via the underlying service at USAP (UDP Service Access Point) to provide the conference service at CSAP (Conference Service Access Point). The CPE has four PDUs: *join-PDU*, *answer-PDU*, *data-PDU* and *leave-PDU*, which can be sent and received according to a number of rules, of which the details are omitted here. Moreover, every CPE is responsible for the administration of two sets, the *potential conference partners* and the *conference partners*. The first is static and contains all users who are allowed to participate in a conference, and the second is dynamic and contains all conference partners (in the form of names and UDP-addresses) that currently participate in the same conference.

Figure 6.2 gives two example instances of behaviour: in (*a*) a *join* service primitive results in sending a *join-PDU*, which is acknowledged by an *answer-PDU*; in (*b*) a *datareq* service primitive leads to a *data-PDU* being sent to all conference partners, which, in turn, invoke a *dataind* primitive.

Formal specifications The protocol has been specified in Promela and in Lotos. Instantiating thePromela-specification with three potential conference users, a model

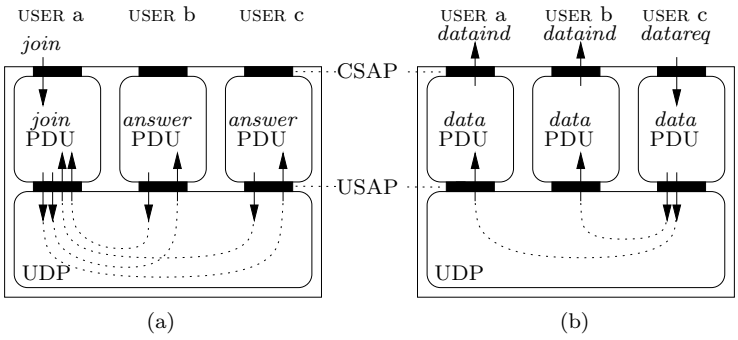


Figure 6.2: The conference protocol

for testing was generated with 122 states and 5 processes. Communication between conference partners was modelled by a set of processes, one for each potential receiver, to allow all possible interleavings between the several sendings of multicast PDUs. For model checking and simulation of the PROMELA model with SPIN [Spi], the user does not only need to provide the behaviour of the system itself but also the behaviour of the system environment. For testing this is not required, see [VT00]. Some PROMELA channels are marked as observable, viz. the ones where observable actions may occur.

Also in the LOTOS specification the number of users has been restricted to three. Moreover, all communication queues have been explicitly modelled.

Conference protocol implementations The conference protocol has been implemented on SUN SPARC workstations using a UNIX-like (SOLARIS) operating system and using the ANSI-C programming language. Furthermore, we used only standard UNIX inter-process and inter-machine communication facilities, such as uni-directional pipes and sockets.

A conference protocol implementation consists of the actual CPE which implements the protocol behaviour and a user-interface on top of it. We require that the user-interface is separated (loosely coupled) from the CPE to isolate the protocol entity; only the CPE is the object of testing. This is realistic because user interfaces are often implemented using dedicated software.

The conference protocol implementation has two interfaces: the CSAP and the USAP. The CSAP interface allows communication between the two UNIX processes, the user-interface and the CPE, and is implemented by two uni-directional pipes. The USAP interface allows communication between the CPE and the underlaying layer UDP, and is implemented by sockets.

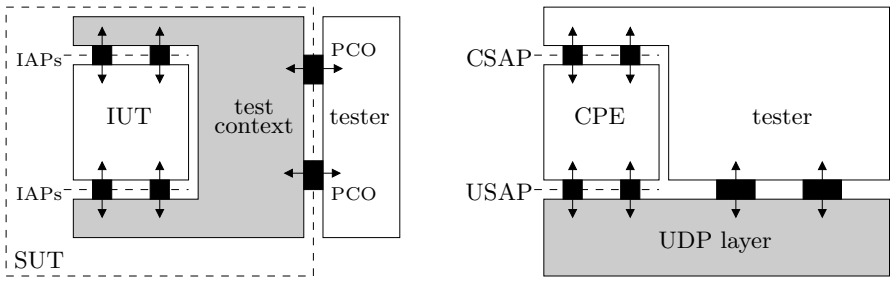
In order to guarantee that a conference protocol entity has knowledge about the potential conference partners the conference protocol entity reads a configuration file during the initialization phase.

Error seeding For our experiment with automatic testing we developed 28 different conference protocol implementations. One of these implementations is correct (at

least, to our knowledge), whereas in 27 of them a single error was injected deliberately. The erroneous implementations can be categorized in three different groups: *No outputs*, *No internal checks* and *No internal updates*. The group *No outputs* contains implementations that forget to send output when they are required to do so. The group *No internal checks* contains implementations that do not check whether the implementations are allowed to participate in the same conference according to the set of potential conference partners and the set of conference partners. The group *No internal updates* contains implementations that do not correctly administrate the set of conference partners.

6.2.2 Test Architecture

For testing a conference protocol entity (CPE) implementation, knowledge about the environment in which it is tested, i.e., the *test architecture*, is essential. A test architecture can (abstractly) be described in terms of the tester – in our case TorX, the implementation under test (IUT) – the CPE –, a test context, the *Points of Control and Observation* (PCOs), and the *Implementation Access Points* (IAPs) [ISO96]. The test context is the environment in which the IUT is embedded and that is present during testing, but that is not the aim of conformance testing. The communication interfaces between the IUT and the test context are defined by IAPs, and the communication interfaces between the test context and TorX are defined by PCOs. The SUT (*System Under Test*) consists of the IUT embedded in its test context. Figure 6.3(a) depicts an abstract test architecture.



(a) Abstract test architecture (b) Test architecture for conference protocol entities

Figure 6.3: Test architecture

Ideally, the tester accesses the CPE directly at its IAPs, both at the CSAP and the USAP level. In our test architecture, which is the same as in [TFPHT96], this is not the case. The tester communicates with the CPE at the USAP via the underlying UDP layer; this UDP layer acts as the test context. Since UDP behaves as an unreliable channel, this complicates the testing process. To avoid this complication we make the assumption that communication via UDP is reliable and that messages are delivered in sequence. This assumption is realistic if we require that the tester and the CPE reside on the same host machine, so that messages exchanged via UDP do not have to travel through the protocol layers below IP but 'bounce back' at IP.

With respect to the IAP at the CSAP interface we already assumed in the previous section that the user interface can be separated from the core CPE. Since the CSAP

interface is implemented by means of pipes the tester therefore has to access the CSAP interface via the pipe mechanism.

Figure 6.3(b) depicts the concrete test architecture. The SUT consists of the CPE together with the reliable UDP service provider. The tester accesses the IAPs at the CSAP level directly, and the IAPs at USAP level via the UDP layer.

Formal model of the test architecture For formal test derivation, a realistic model of the behavioural properties of the complete SUT is required, i.e., the CPE and the test context, as well as the communication interfaces (IAPs and PCOs). The formal model of the CPE is based on the formal specifications, see Section 6.2.1. Using our assumption that the tester and the CPE reside on the same host, the test context (i.e., the UDP layer) acts as a reliable channel that provides in-sequence delivery. This can be modelled by two unbounded first-in/first-out (FIFO) queues, one for message transfer from tester to CPE, and one vice versa. The CSAP interface is implemented by means of pipes, which essentially behave like bounded first-in/first-out (FIFO) buffers. Under the assumption that a pipe is never 'overloaded', this can also be modelled as an unbounded FIFO queue. The USAP interface is implemented by means of sockets. Sockets can also be modelled, just as pipes, by unbounded FIFO queues. Finally, the number of communicating peer entities of the CPE, i.e., the set of potential conference partners, has been fixed in the test architecture to two. Figure 6.4 visualizes the complete formal models of the SUT as they were written in PROMELA and LOTOS.

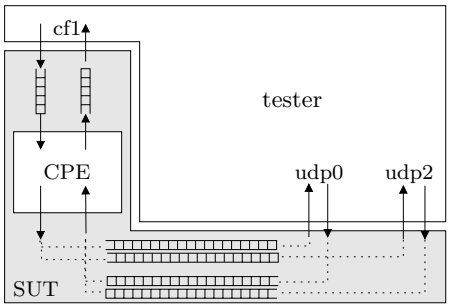


Figure 6.4: Formal model of the SUT

6.2.3 Testing Activities

This section describes our testing activities for the PROMELA specification with on-the-fly test generation and execution, using the **ioco**-correctness criterion (definition 4.5.1).

We started with initial experiments to identify errors in the specification and to test the (specification language specific) en/decoding functions of the tester. Once we had sufficient confidence in the specification and en/decoding functions, we tested the (assumed to be) correct implementation, after which the 27 erroneous mutants were tested by people who did not know which errors had been introduced in these mutants.

Initial experiments We started with repeatedly running TorX in automatic mode, each time with a different seed for the random number generator, until either a depth of 500 steps was reached or an inconsistency between specification and implementation was detected (i.e., **fail**, usually after some 30 to 70 steps). This uncovered some errors in both the implementation and the specification, which were repaired. In addition, we have run TorX in user-guided, manual mode to explore specific scenarios and to analyse failures that were found in fully automatic mode.

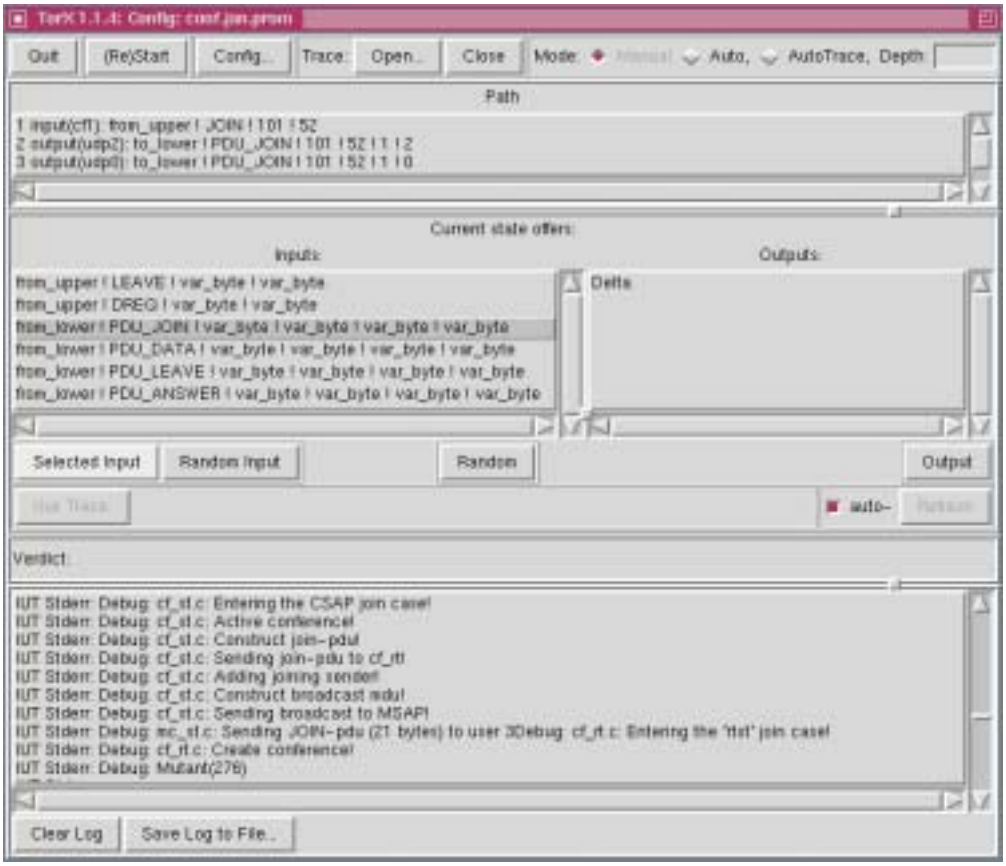


Figure 6.5: TorX graphical user interface – selecting input

Manual guidance Figure 6.5 shows the graphical user interface of TorX; this is the user interface that we used for user-guided, manual mode testing. From top to bottom, the most important elements in it are the following. At the top, the *Path* pane shows the test events that have been executed so far. Below it, the *Current state offers* pane shows the possible inputs and outputs at the current state. Almost at the bottom, the *Verdict* pane will show us the verdict. Finally, the bottom pane shows diagnostic output from TorX, and diagnostic messages from the SUT.

In the *Path* pane we see the following scenario. In test event 1, the conference user (played by TorX) joins conference 52 with user-id 101, by issuing a *join* service primitive at PCO cf1. The SUT (at address 1) informs the two other potential conference partners at PCO udp2 (at address 2) and PCO udp0 (at address 0) in test event 2 resp. test event 3 using *join-PDUs*. The *Current state offers* pane shows the possible

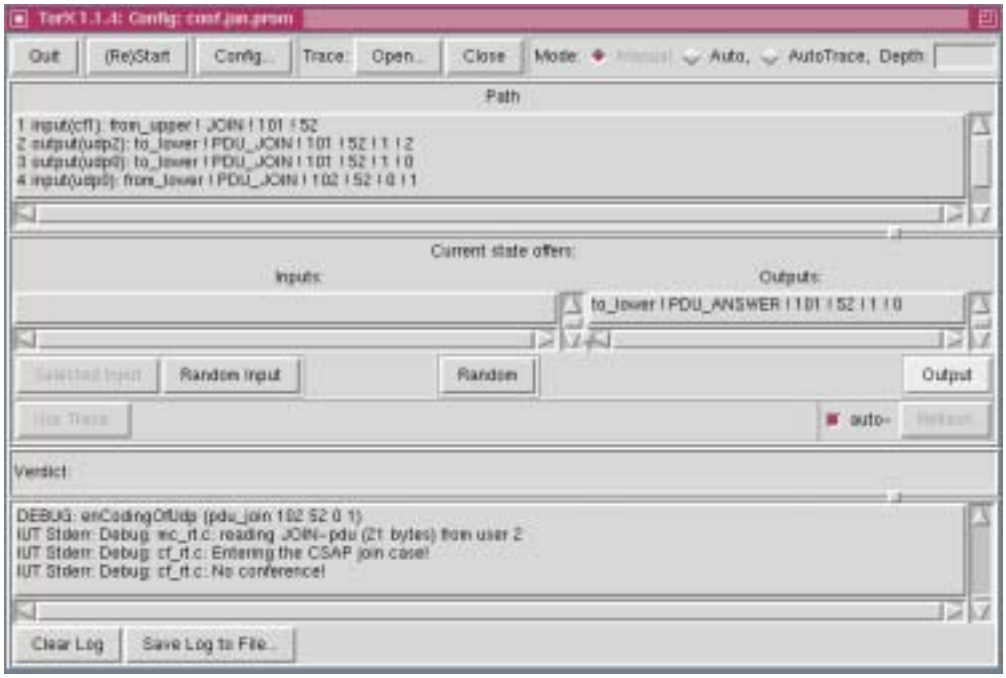


Figure 6.6: TorX graphical user interface – observing output

inputs and the expected outputs (output *Delta* means *quiescence*, see Chapter 3). The input action that we are about to select is highlighted: we will let a conference partner join a conference as well. The tester will choose values for the variables shown in the selected input event, if we do not supply values ourselves.

Figure 6.6 shows the TorX window after selecting the Selected Input button. Test event 4 in the *Path* pane shows that now the conference partner at PCO udp0 has joined conference 52 as well, using user-id 102, by sending a *join-PDU* (from address 0) to the SUT (at address 1). The list of *outputs* now shows that the SUT (at address 1) should respond by sending an *answer-PDU* containing its own user-id and conference-id to address 0, i.e., to PCO udp0. We will now select the Output button to ask TorX for an observation.

Figure 6.7 shows the TorX window after selecting the Output button to observe an output. The presence of test event 5, *Quiescence*, shows that the tester did not receive anything from the SUT, which is not allowed (there is no *Delta* in the list of outputs), and therefore TorX issues a **fail** verdict. Indeed, in this example we tested one of the mutants, in particular, the one that does not remember that it has joined a conference, and therefore does not respond to incoming *join-PDUs*.

In a separate window, TorX keeps an up-to-date message sequence chart (MSC) of the test run. This message sequence chart shows the messages interchanged between the SUT (*iut*) and each of the PCO's. A separate line represents the 'target' of quiescence. The message sequence chart for the test run described above is shown in Figure 6.8.

Long-running experiment Once we had sufficient confidence in the quality of the specification and implementation we repeated the previous 'automatic mode' experi-

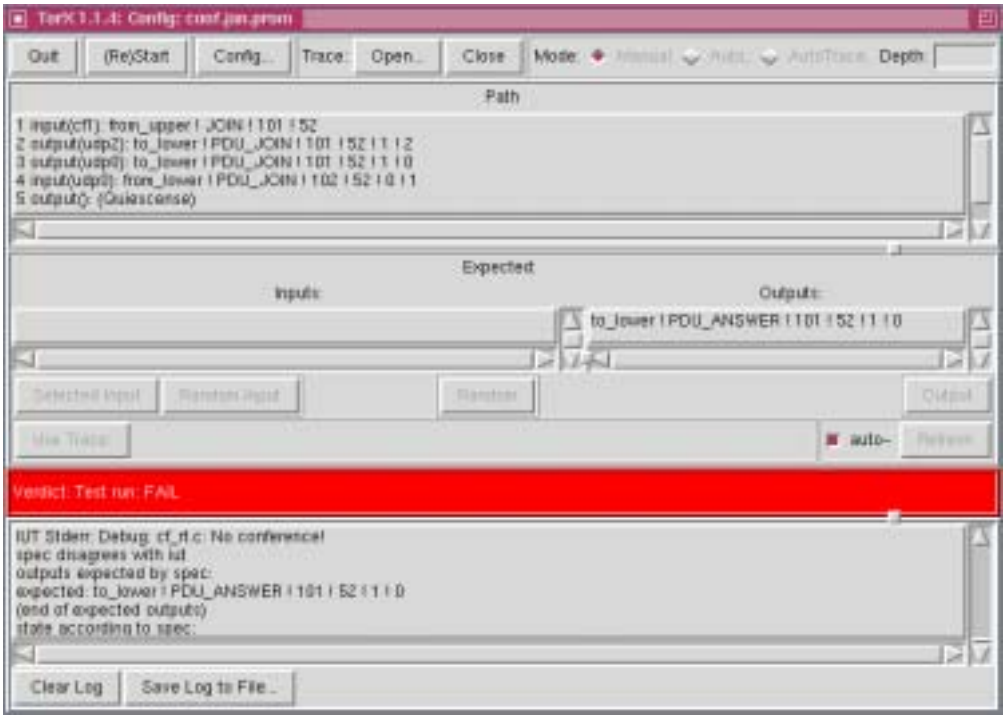


Figure 6.7: TorX graphical user interface – final verdict

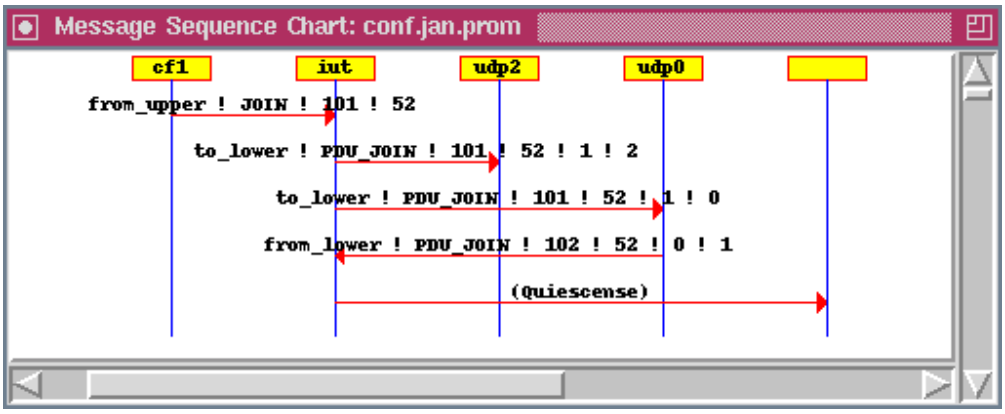


Figure 6.8: TorX Message Sequence Chart

ment, but now we tried to execute as many test steps as possible. The longest trace we were able to execute consisted of 450,000 steps and took 400 Mb of memory. On average the execution time was about 1.1 steps per second.

Mutants detection To test the error-detection capabilities of our tester we repeatedly ran TorX in automatic mode for a depth of 500 steps, each time with a different seed for the random number generator, on the 27 mutants. The tester was able to detect 25 of them. The number of test events in the shortest test runs that detected the mutants ranged from 2 to 38; on average 18 test events were needed. The two mutants that could not be detected accept PDUs from any source – they do not check whether an incoming PDU comes from a potential conference parter. This is not explicitly modeled in our Promela specification, and therefore these mutants are **ioco**-correct with respect to the Promela specification, which is why we cannot detect them.

Other formalisms We have repeated the experiments described above with specifications in LOTOS and SDL, as described in [BFV+99]. In the case of LOTOS, we used the same fully-automatic on-the-fly test-derivation and execution approach, giving us the same results as for Promela, but needing more processing time and consuming more memory. The main reason for this is that the Promela-based tester uses the memory-efficient internal data representations and hashing techniques to remember the result of unfoldings from Spin [Hol91, Spi]. In the case of SDL, we used a user-guided test-derivation approach, after which the test-cases were automatically executed. Here we were not able to detect all mutants, but this may be due to the limited number of test cases that we derived; by deriving more test cases we will likely be able to detect more mutants.

Characteristics of test cases Which test cases are generated by TorX only depends on the seed of the random number generator with which TorX is started, and the (possibly non-deterministic) outputs of the SUT. From one seed to another, the number of test events needed to trigger an error may vary significantly. For the conference protocol, for one seed it took only two test events to detect a mutant, and for another seed it took 10 test events to detect the same mutant. For another mutant the differences were even more extreme: for one seed it could be detected in 24 test events, for another seed it took 498 steps. Still, as effectively in all test runs all mutants were (finally) detected, the results seem to indicate that if we run TorX sufficiently often, with varying seeds, and let it run long enough, then all errors are found. This is, unfortunately, currently also the only indication of the coverage and error-detecting capability that we have.

Logfile analysis During a test run, a log is kept to allow analysis of the test run. Such a log contains not only behavioural information to allow analysis of the test run, like tester configuration information, and the executed test events in concrete and in abstract form, but also information that allows analysis of the tester itself, like, for each test event, a time-stamp, the memory usage of the computation of the test event, and a number of other statistics from the test derivation module. In case of a **fail** verdict, the expected outputs are included in the log. TorX is able to rerun a log created in a previous test run. This may fail if the SUT behaves nondeterministically, i.e., if the order in which outputs are observed during the rerun differs from the order

in which they are present in the log. This limitation may show up, for example, for the *join-PDUs* in test events 2 and 3 in Figure 6.5.

Comparison with traditional testing In traditional testing the number of test events needed to trigger an error will quite likely be smaller than for TorX, thanks to the human guidance that will lead to 'efficient' test cases, whereas TorX may 'wander' around the error for quite a while until finally triggering it. On the other hand, to generate a new test case with TorX it is sufficient to invoke it once more with a so far untried seed, whereas approaches that are based on manual production of test cases need considerably more manual effort to derive more test cases. The main investment needed to use TorX lies in making the specification, and connecting TorX to the SUT. Once that has been arranged, the testing itself is just a case of running TorX often and long enough (and, of course, analysing the test logs).

6.3 Industrial Applications

Formal testing with TorX has been applied in the following industrial case studies:

- The Philips EasyLink protocol for down-loading channel pre-sets from a television set to a video recorder was tested with TorX with respect to a Promela specification of behaviour. One fault was detected which had slipped through all traditional, manual testing [BFHV01].
- An Access Network V5.1 protocol entity of Lucent Technologies was tested. Via reverse engineering a Lotos specification was developed which was used as the basis for testing.
- A software module of the component-wise developed new Cell-Broadcast-Centre of CMG was tested with TorX based on a Lotos specification and an interface description in IDL. A comparison with the manually developed test suite was favourable for TorX [Chr01].
- Interpay's Payment Box of the *Rekeningrijden* system was tested with TorX and Promela. During validation of the Promela model a design error was found. In the testing phase some strange behaviour was detected during long tests ($\geq$ 70,000 test events), but no proof of non-conformance was found in this product which had already been tested using all conventional testing phases [VBF02].

Bibliography

[Abr87] S. Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, 53(3):225–241, 1987.

[Ald90] R. Alderden. COOPER, the compositional construction of a canonical tester. In S.T. Vuong, editor, *FORTE'89*, pages 13–17. North-Holland, 1990.

[Bae86] J.C.M. Baeten. *Procesalgebra*. Kluwer, 1986. (In Dutch).

[BAL+90] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A formal approach to conformance testing. In J. de Meer, L. Mackert, and W. Effelsberg, editors, *Second Int. Workshop on Protocol Test Systems*, pages 349–363. North-Holland, 1990. Also: Memorandum INF-89-45, University of Twente, The Netherlands.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990. Second Edition.

[Ber91] G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky and T.S.E. Maibaum, editors, *TAPSOFT'91, Volume 2*, pages 99–119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.

[BFHV01] A. Belinfante, J. Feenstra, L. Heerink, and R.G. de Vries. Specification Based Formal Testing: The EasyLink Case Study. In *Progress 2001 – 2^{nd} Workshop on Embedded Systems*, pages 73–82, Utrecht, The Netherlands, October 18 2001. STW Technology Foundation.

[BFV+99] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal Test Automation: A Simple Experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Int. Workshop on Testing of Communicating Systems 12*, pages 179–196. Kluwer Academic Publishers, 1999.

[BHT97] E. Brinksma, L. Heerink, and J. Tretmans. Developments in testing transition systems. In M. Kim, S. Kang, and K. Hong, editors, *Int. Workshop on Testing of Communicating Systems 10*, pages 143–166. Chapman & Hall, 1997.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[Bri87] E. Brinksma. On the existence of canonical testers. Memorandum INF-87-5, University of Twente, Enschede, The Netherlands, 1987.

[Bri88] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 63–74. North-Holland, 1988. Also: Memorandum INF-88-19, University of Twente, The Netherlands.

[Bri93] E. Brinksma. On the coverage of partial validations. In M. Nivat, C.M.I. Rattray, T. Rus, and G. Scollo, editors, *AMAST'93*, pages 247–254. BCS-FACS Workshops in Computing Series, Springer-Verlag, 1993.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In G. von Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing, and Verification VI*, pages 349–360. North-Holland, 1987.

[BT01] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *Modeling and Verification of Parallel Processes – 4^{th} Summer School MOVEP 2000*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer-Verlag, 2001.

[CCI92] CCITT. *Specification and Description Language (SDL)*. Recommendation Z.100. ITU-T General Secretariat, Geneve, Switzerland, 1992.

[CFP94] A.R. Cavalli, J.P. Favreau, and M. Phalippou. Formal methods in conformance testing: Results and perspectives. In O. Rafiq, editor, *Int. Workshop on Protocol Test Systems VI*, number C-19 in IFIP Transactions, pages 3–17. North-Holland, 1994.

[CG97] O. Charles and R. Groz. Basing Test Coverage on a Formalization of Test Hypotheses. In M. Kim, S. Kang, and K. Hong, editors, *Int. Workshop on Testing of Communicating Systems 10*, pages 109–124. Chapman & Hall, 1997.

[CGPT96] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking test generation with verification techniques. In A. Cavalli and S. Budkowski, editors, *Eight Int. Workshop on Protocol Test Systems*. Chapman & Hall, 1996.

[Chr01] P. Christian. Specification Based Testing with IDL and Formal Methods: A Case Study in Test Automation. Master's thesis, University of Twente, Enschede, The Netherlands, 2001.

[CJRZ01] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A Tool for Generating Symbolic Test Programs and Oracles from Operational Specifications. In *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

[Cla96] M. Clatin. Manuel d'Utilisation de TVEDA V3. User Manual LAA/EIA-/EVP/109, France Télécom CNET, Lannion, France, 1996.

[CV97] J.A. Curgus and S.T. Vuong. Sensitivity analysis of the metric based test selection. In M. Kim, S. Kang, and K. Hong, editors, *Int. Workshop on Testing of Communicating Systems 10*, pages 200–219. Chapman & Hall, 1997.

[DAV93] K. Drira, P. Azéma, and F. 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*, number C-16 in IFIP Transactions. North-Holland, 1993.

[DBRV+00] L. Du Bousquet, S. Ramangalshy, C. Viho, A. Belinfante, and R.G. de Vries. Formal Test Automation: The Conference Protocol with TGV/ TORX. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *TestCom 2000*, pages 221–228. Kluwer Academic Publishers, 2000.

[DN87] R. De Nicola. Extensional equivalences for transition systems. *Theoretical Computer Science*, 24:211–237, 1987.

[DNH84] R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DNS95] R. De Nicola and R. Segala. A process algebraic view of Input/Output Automata. *Theoretical Computer Science*, 138:391–423, 1995.

[Doo91] P. Doornbosch. Test Derivation for Full LOTOS. Memorandum INF-91-51, University of Twente, Enschede, The Netherlands, 1991. Master's Thesis.

[Dri92] K. Drira. *Transformation et Composition de Graphes de Refus: Analyse de la Testabilité*. PhD thesis, Laboratoire d'Automatique et d'Analyse des Systemes du CNRS, Toulouse, France, 1992.

[Eer87] H. Eertink. The implementation of a test derivation algorithm. Memorandum INF-87-36, University of Twente, Enschede, The Netherlands, 1987.

[FG99] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley, 1999.

[FGST02] L.M.G. Feijs, N. Goga, Mauw S., and J. Tretmans. Test Selection, Trace Distance and Heuristics. In H. König, I. Schieferdecker, and A. Wolisz, editors, *TestCom 2002*. Kluwer Academic Publishers, 2002. To appear.

[FJJV96] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification CAV'96*. Lecture Notes in Computer Science 1102, Springer-Verlag, 1996.

[FJJV97] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming – Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1–2):123–146, 1997.

[FP95] L. Ferreira Pires. Protocol implementation: Manual for practical exercises 1995/1996. Lecture notes, University of Twente, Enschede, The Netherlands, August 1995.

[Gar98] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In B. Steffen, editor, *Fourth Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 68–84. Lecture Notes in Computer Science 1384, Springer-Verlag, 1998.

[Gau95] M.-C. Gaudel. Testing can be formal, too. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, pages 82–96. Lecture Notes in Computer Science 915, Springer-Verlag, 1995.

[GJ98] M.-C. Gaudel and P.R. James. Testing algebraic data types and processes: A unifying theory. In J.F. Groote, B. Luttik, and J. van Wamel, editors, *Third Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS'98)*, pages 215–230, Amsterdam, The Netherlands, 1998. CWI.

[Gla90] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90*, Lecture Notes in Computer Science 458, pages 278–297. Springer-Verlag, 1990.

[Gla93] R.J. van Glabbeek. The linear time – branching time spectrum II (The semantics of sequential systems with silent moves). In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 66–81. Springer-Verlag, 1993.

[GSDD97] J. Grabowski, R. Scheurer, Z.R. Dai, and Hogrefe. D. Applying SaMsTaG to the B-ISDN protocol SSCOP. In M. Kim, S. Kang, and K. Hong, editors, *Int. Workshop on Testing of Communicating Systems 10*, pages 397–415. Chapman & Hall, 1997.

[GWT98] W. Geurts, K. Wijbrans, and J. Tretmans. Testing and formal methods — BOS project case study. In *EuroSTAR'98: 6^{th} European Int. Conference on Software Testing, Analysis & Review*, pages 215–229, Munich, Germany, November 30 – December 1 1998. Aimware, Mervue, Galway, Ireland.

[Hee98] L. Heerink. *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.

[Hen88] M. Hennessy. *Algebraic Theory of Processes*. Foundations of Computing Series. The MIT Press, 1988.

[HFT00] L. Heerink, J. Feenstra, and J. Tretmans. Formal Test Automation: The Conference Protocol with PHACT. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *Testing of Communicating Systems – Procs. of TestCom 2000*, pages 211–220. Kluwer Academic Publishers, 2000.

[Hie97] R.M. Hierons. Testing from a Z Specification. *Software Testing, Verification and Reliability*, 7:19–33, 1997.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall Inc., 1991.

[HST96] D. Hogrefe, Heymer. S., and J. Tretmans. Report on the standardization project "Formal Methods in Conformance Testing". In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *Int. Workshop on Testing of Communicating Systems 9*, pages 289–298. Chapman & Hall, 1996.

[HT96] L. Heerink and J. Tretmans. Formal methods in conformance testing: A probabilistic refinement. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *Int. Workshop on Testing of Communicating Systems 9*, pages 261–276. Chapman & Hall, 1996.

[HT97] L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Desciption Techniques and Protocol Specification, Testing and Verification FORTE X /PSTV XVII '97*, pages 23–38. Chapman & Hall, 1997.

[HT99] J. He and K.J. Turner. Protocol-Inspired Hardware Testing. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Int. Workshop on Testing of Communicating Systems 12*, pages 131–147. Kluwer Academic Publishers, 1999.

[ISO89] ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. International Standard IS-8807. ISO, Geneve, 1989.

[ISO91] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*. International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.

[ISO96] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing*. Committee Draft CD 13245-1, ITU-T Recommendation Z.500. ISO – ITU-T, Geneve, 1996.

[JJKV98] C. Jard, T. Jéron, H. Kahlouche, and C. Viho. Towards Automatic Distribution of Testers for Distributed Conformance Testing. In S. Budkowski, A. Cavalli, and Najm E., editors, *Formal Desciption Techniques and Protocol Specification, Testing and Verification FORTE X /PSTV XVII '98*, pages 353–368. Kluwer Academic Publishers, 1998.

[JJM00] C. Jard, T. Jéron, and P. Morel. Verification of Test Suites. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *Testing of Communicating Systems – Procs. of TestCom 2000*, pages 3–18. Kluwer Academic Publishers, 2000.

[JM99] T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification CAV'99*, pages 108–121. Lecture Notes in Computer Science 1633, Springer-Verlag, 1999.

[KJG99] A. Kerbrat, T. Jéron, and R. Groz. Automated Test Generation from SDL Specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99, The Next Millennium – Proceedings of the* 9^{th} *SDL Forum*, pages 135–152. Elsevier Science, 1999.

[KP99] T. Koomen and M. Pol. *Test Process Improvement – A Practical Step-by-Step Guide to Structured Testing*. Addison-Wesley, 1999.

[KVZ98] H. Kahlouche, C. Viho, and M. Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In A. Petrenko and N. Yevtushenko, editors, *Int. Workshop on Testing of Communicating Systems 11*, pages 211–226. Kluwer Academic Publishers, 1998.

[Lan90] R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 87–98. North-Holland, 1990.

[Led92] G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25(1):23–41, 1992.

[LGA96] P. Le Gall and A. Arnould. Formal specifications and test: Correctness and oracle. In O.-J. Dahl, O. Owe, and M. Haveraaen, editors, 11^{th} *ADT Workshop*. Lecture Notes in Computer Science, Springer-Verlag, 1996.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989. Also: Technical Report MIT/LCS/TM-373 (TM-351 revised), Massachusetts Institute of Technology, Cambridge, U.S.A., 1988.

[LY96] D. Lee and M. Yannakakis. Principles and methods for testing finite state machines – a survey. *The Proceedings of the IEEE*, 84(8):1090–1123., August 1996.

[Mar95] B. Marre. LOFT: A tool for assisting selection of test data sets from algebraic specifications. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, pages 799–800. Lecture Notes in Computer Science 915, Springer-Verlag, 1995.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mye79] G.J. Myers. *The Art of Software Testing*. John Wiley & Sons Inc., 1979.

[NS01] B. Nielsen and A. Skou. Automated Test Generation from Timed Automata. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 343–357. Springer-Verlag, 2001.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5th GI Conference*, pages 167–183. Lecture Notes in Computer Science 104, Springer-Verlag, 1981.

[PF90] D. H. Pitt and D. Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, 1990.

[Pha94a] M. Phalippou. Executable testers. In O. Rafiq, editor, *Int. Workshop on Protocol Test Systems VI*, number C-19 in IFIP Transactions, pages 35–50. North-Holland, 1994.

[Pha94b] M. Phalippou. *Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties*. PhD thesis, L'Université de Bordeaux I, France, 1994.

[Phi87] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(2):241–284, 1987.

[Pro] Project Consortium Côte de Resyste. Conference Protocol Case Study. URL: `http://fmt.cs.utwente.nl/ConfCase`.

[PS97] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Jounal*, 19:53–77, 1997.

[PV98] M. Pol and E. van Veenendaal. *Structured Testing of Information Systems – An Introduction to TMap*. Kluwer, The Netherlands, 1998.

[PWC95] M.C. Paulk, C.V. Weber, and B. Curtis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. SEI Series in Software Engineering. Addison-Wesley, 1995.

[Ray87] D. Rayner. OSI conformance testing. *Computer Networks and ISDN Systems*, 14:79–98, 1987.

[Ray97] D. Rayner. Future directions for protocol testing, learning the lessons from the past. In M. Kim, S. Kang, and K. Hong, editors, *Int. Workshop on Testing of Communicating Systems 10*, pages 3–17. Chapman & Hall, 1997.

[Seg93] R. Segala. Quiescence, fairness, testing, and the notion of implementation. In E. Best, editor, *CONCUR'93*, pages 324–338. Lecture Notes in Computer Science 715, Springer-Verlag, 1993.

[SEK+98] M. Schmitt, A. Ek, B. Koch, J. Grabowski, and D. Hogrefe. – AUTOLINK – Putting SDL-based Test Generation into Practice. In A. Petrenko and N. Yevtushenko, editors, *Int. Workshop on Testing of Communicating Systems 11*, pages 227–243. Kluwer Academic Publishers, 1998.

[Sit00] M. Siteur. *Testen met Testtools – Slapend Werken*. Academic Service, 2000. url: `http://www.siteur.myweb.nl/`. (In Dutch).

[Spi] SPIN – On-the-Fly, LTL Model Checking with SPIN. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[STW96] Dutch Technology Foundation STW. *Côte de Resyste* – COnformance TEsting of REactive SYSTEms. Project proposal STW TIF.4111, University of Twente, Eindhoven University of Technology, Philips Research Laboratories, KPN Research, Utrecht, The Netherlands, 1996. `http://fmt.cs.utwente.nl/CdR`.

[Tan97] Q. Tan. *On Conformance Testing of Systems Communicating by Rendezvous*. PhD thesis, Université de Montréal, Montréal, Canada, 1997.

[TB99] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland. Also: Technical Report TR-CTIT-17, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[TFPHT96] R. Terpstra, L. Ferreira Pires, L. Heerink, and J. Tretmans. Testing theory in practice: A simple experiment. In T. Kapus and Z. Brezočnik, editors, *COST 247 Int. Workshop on Applied Formal Methods in System Design*, pages 168–183, Maribor, Slovenia, 1996. University of Maribor. Also: Technical Report No. 96-21, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[TKB92] J. Tretmans, P. Kars, and E. Brinksma. Protocol conformance testing: A formal perspective on ISO IS-9646. In J. Kroon, R. J. Heijink, and E. Brinksma, editors, *Fourth Int. Workshop on Protocol Test Systems*, number C-3 in IFIP Transactions, pages 131–142. North-Holland, 1992. Extended abstract of Memorandum INF-91-32, University of Twente, The Netherlands, 1991.

[TPB96] Q.M. Tan, A. Petrenko, and G. von Bochmann. Modeling Basic LOTOS by FSMs for conformance testing. In P. Dembiński and M. Średniawa, editors, *Protocol Specification, Testing, and Verification XV*, pages 137–152. IFIP WG6.1, Chapman & Hall, 1996. Also: publication # 958, Université de Montréal, Département d'Informatique et de Recherche Opérationnelle.

[Tre90] J. Tretmans. Test case derivation from LOTOS specifications. In S.T. Vuong, editor, *FORTE'89*, pages 345–359. North-Holland, 1990. Also: Memorandum INF-90-21, University of Twente, The Netherlands.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[Tre94] J. Tretmans. A formal approach to conformance testing. In O. Rafiq, editor, *Int. Workshop on Protocol Test Systems VI*, number C-19 in IFIP Transactions, pages 257–276. North-Holland, 1994.

[Tre96a] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[Tre96b] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996. Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[Tre99] J. Tretmans. Testing Concurrent Systems: A Formal Approach. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.

[TV92] J. Tretmans and L. Verhaard. A queue model relating synchronous and asynchronous communication. In R.J. Linn and M.Ü. Uyar, editors, *Protocol Specification, Testing, and Verification XII*, number C-8 in IFIP Transactions, pages 131–145. North-Holland, 1992. Extended abstract of Memorandum INF-92-04, University of Twente, Enschede, The Netherlands, 1992, and Internal Report, TFL RR 1992-1, TFL, Hørsholm, Denmark.

[TWC01] J. Tretmans, K. Wijbrans, and M. Chaudron. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System – Revisiting Seven Myths of Formal Methods. *Formal Methods in System Design*, 19(2):195–215, 2001.

[Vaa91] F. Vaandrager. On the relationship between process algebra and Input/Output Automata. In *Logic in Computer Science*, pages 387–398. Sixth Annual IEEE Symposium, IEEE Computer Society Press, 1991.

[VBF02] R.G. de Vries, A. Belinfante, and J. Feenstra. Automated Testing in Practice: The Highway Tolling System. In H. König, I. Schieferdecker, and A. Wolisz, editors, *TestCom 2002*. Kluwer Academic Publisher, 2002.

[VT00] R.G. de Vries and J. Tretmans. On-the-Fly Conformance Testing using Spin. *Software Tools for Technology Transfer*, 2(4):382–393, 2000.

[VTKB93] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In G. von Bochmann, R. Dssouli, and A. Das, editors, *Fifth Int. Workshop on Protocol Test Systems*, IFIP Transactions. North-Holland, 1993. Also: Memorandum INF-93-03, University of Twente, The Netherlands.

[Wez90] C. D. Wezeman. The CO-OP method for compositional derivation of conformance testers. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 145–158. North-Holland, 1990.

Part II

Articles

Chapter 1

What is Software Testing? And Why is it So Hard?

J.A. Whittaker,
What is software testing? And why is it so hard?,
IEEE Software, Vol. 17, Nr. 1 (January/February), 2000, pages 70–79.
IEEE Computer Society, 2000. ISSN: 0740-7459.
`http://computer.org/software/so2000/pdf/s1070.pdf`.

Chapter 2

Test-Case Design

Glenford J. Myers,
The Art of Software Testing,
Chapter 4: *Test-Case Design*; pages 36–55.
John Wiley & Sons, 1979. ISBN: 0-471-04328-1.

Chapter 3

Formal Methods for Test Sequence Generation

H. Ural,
Formal Methods for Test Sequence Generation,
Computer Communications Journal, Vol. 15, Nr. 5, 1992, pages 311-317,
(311–325 for the complete article).
Butterworth-Heinemann. ISSN: 0140-3664.

Chapter 4

An Overview of OSI Conformance Testing

Translated and adapted from:

J. Tretmans, J. van de Lagemaat,
Conformance Testen,
Handboek Telematica, Vol. II, pages 4400 1–19,
Samson, 1991.

Chapter 5

Getting Automated Testing under Control

H. Buwalda, M. Kasdorp,
Getting Automated Testing under Control,
Software Testing & Quality Engineering,
Vol. 1, Nr. 6 (November/December), 1999, pages 38–44.
An SQE Publication; `www.stqemagazine.com`.

Chapter 6

Test Management Approach

T. Koomen, M. Pol,
Test Process Improvement – A Practical Step-by-Step Guide to Structured Testing,
Appendix B: *Summary of TMap*; pages 173–187.
Addison-Wesley, 1999. ISBN 0-201-59624-5.

Chapter 7

Test Tools

T. Koomen, M. Pol,
Test Process Improvement – A Practical Step-by-Step Guide to Structured Testing,
Appendix B: *Test Tools*; pages 189–197.
Addison-Wesley, 1999. ISBN 0-201-59624-5.