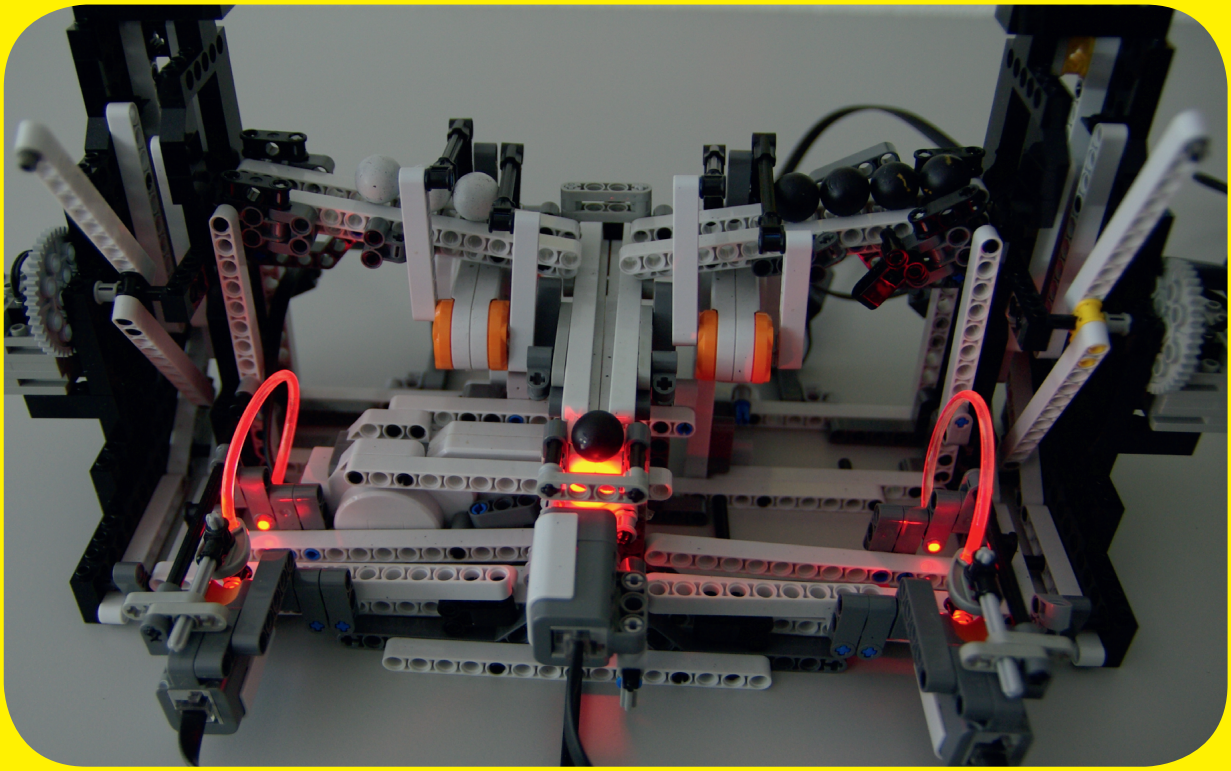
Belinfante

Ede

Frits

Axel



JTorX:

Exploring

Model-Based

Testing

JTorX: Exploring Model-Based Testing

Axel Belinfante

Graduation committee:

Chairman:	Prof. dr. Peter M.G. Apers
Promotors:	Prof. dr. Jaco C. van de Pol
	Prof. dr. ir. Arend Rensink

Members:

Prof. dr. ir. Aiko Pras	Universiteit Twente
Prof. dr. ir. Roel J. Wieringa	Universiteit Twente
Prof. dr. Alexander Pretschner	TU München
dr. Thierry Jéron	IRISA / INRIA Rennes
dr. Jan G. Tretmans	Radboud Universiteit Nijmegen
Prof. dr. Ed Brinksma	Universiteit Twente (acting chairman)

CTIT Ph.D. Thesis Series No. 14-319
Centre for Telematics and Information Technology
University of Twente
P.O. Box 217 – 7500 AE Enschede, NL

IPA Dissertation Series No. 2014-09
The work in this thesis has be carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 978-90-365-3707-0
ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. 14-319)
DOI 10.3990/1.9789036537070
URL `http://dx.doi.org/10.3990/1.9789036537070`

Cover picture: the Lego Mindstorms ball sorter, developed to make model-based testing with JTorX tangible. This sorter was designed and constructed by Arjan Snippe, and then enhanced by Mathijs van der Werff. Spine center picture: a home-built revolution [Rev] kite; spine edge pictures: home-built Vluis [Buu95] kites.

Edited with Wily, Acme SAC, and Plan9port acme (and occasionally vi).
Typeset with LaTeX
Printed by Ipskamp Drukkers Enschede

JTORX: EXPLORING MODEL-BASED TESTING

PROEFONTWERP

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 18 september 2014 om 14:45 uur

door

Axel Frits Ede Belinfante

geboren op 4 juli 1964
te Alkmaar

This dissertation has been approved by:

Prof. dr. Jaco van de Pol (promotor)
Prof. dr. ir. Arend Rensink (promotor)

Aan Sandra, Esther en Ede

Acknowledgements

Many people were, in some way or another, helpful in making this thesis happen, you probably know who you are, and I would like to thank all of you! I will try to mention your names, below, but if your name by accident doesn't appear here—which might easily happen, given that the work described in this thesis took place over a significant number of years— please don't feel left out, your help is appreciated nevertheless! :-)

To start, I would like to thank *testman* Jan for inviting me, so many years ago, to do some work on a prototype testing tool—in the course of the Côte-de-Resyste project, this prototype eventually evolved to what became known as TorX. Then, of course, it is time to thank Ed for helping me to bite the bullet and finally start working on a thesis... neither of us realizing that it would take so many years to complete it. Having Ed, the initial supervisor of this thesis, as chairman of the defense ceremony makes it very special for me: now the words "Mister Rector", typically spoken to a "replacement", will actually address the R.M. (in the words of Rom) of this university! Next, I would like to thank Jaco for allowing me to continue work on the thesis, and, in particular, in helping me to find a way to describe my thoughts about the symbolic part. Finally, I would like to thank Arend for his help in deciding on the overall structure of the thesis, fiddling with the formal notation, and in helping to polish my writing, and making it actually happen. I also want to thank the members of the committee, Aiko, Roel, and Alexander, Jan and Thierry in particular, for their (extensive) feedback. Thank you all: without you, this thesis would never have materialized.

Next, in order of appearance, more or less...

I would like to thank all members of the Côte-de-Resyste project: Loe, Sjouke, Nicolae, Lex, Ron, Erik, Arjan, Ed, Jan, Henk, and of course my room mates René and Jan. Thank you, other room mates: Henk, Han, Laura, André, Vincent, Jeroen, Mohsin, Pedro, Andreas, Joost-Pieter, Theo.

I would like to thank the people at Inria/Irisa Rennes that made our "van Gogh" visits to Rennes so enjoyable, in particular Thierry, Vlad, Claude, Solofo, Lydie, Séverine, and César.

Thank you Albert, for some early adapter-related discussions.

Thank you, members of the Plan 9 community, for inspiration and nice meetings. A special thank you to Sape, and to those of you who came to the Twente9con meeting.

Thank you, Jan-Friso and Muck for your super-fast creation of the lps2torx

component in the mCRL2 toolset during a one-day visit to Twente, and Jeroen for extending it with functionality to measure coverage of lps summands. Thank you, Stefan, for your help creating the lps2torx component for LTSmin.

Thank you, Machiel and Wouter, for your help with the "Oosterschelde Storm Surge Barrier" case (and, Machiel, for all the rest).

Thank you, all participants of the GI/Dagstuhl Research Seminar "Model-based Testing of Reactive Systems", and in particular the co-authors of our chapter: Lars and Christian.

Thank you, "testing group" members: Marielle, Machiel, Laura, Henrik.

Thank you Lars, for developing STSimulator; Henrik, for developing ta2torx; Sabrina, for the discussions and brainstorming about data and time. Thank you David, for your lazy-otf work, and enhancements of STSimulator and JTorX.

Thank you, Jeroen, voor Spex. Thank you, Marten, for your great internship at Neopost, as one of the first serious users of JTorX, and Marielle, for you contributions to the (journal) paper that we wrote about it—I learned a lot in the process. Thank you Thijs and Jeroen, for using JTorX at ProRail and Panalytical, and providing feedback about it. To all other students—I am not going to name you here individually: thank you!

Thank you Arjan, for developing our nice Lego ball-sorter demo, and Mathijs, for improving its robustness, and making it such that we can test "forever". Thank you Mark, for the rivercrossing implementation, and your other contributions to the testing techniques lab class (and for being one of the first users of puptol; regarding puptol thanks also go to Freark, Dennis and Stefan). Thank you, Boni, for your kind permission to use your mathcats.com images for the rivercrossing visualisation/animation. Thank you, Andrew, for making your nice Lego elevator design available on the web: combined with Jaco's lift model it gave us yet another nice JTorX demo.

Thank you, Tangram project members—I have nice memories of a cooking event! Thank you, Quasimodo project members, for the various nice and interesting meetings. Thank you, Marcel, Frits and Bert, Jan and Feng, for your help with the "Myrianed Wireless Sensor Node" case. Thank you, Kai and Hernan, for the opportunity to try the STS support of JTorX on your case study.

Thank you all past and present members of the FMT (and TIOS, and IPS)—in particular those of you with whom I worked together or otherwise spent time with... I'm not going to name you all, but will of course make an exception for Joke and Jeanette.

A special thanks to those kite-flyers and kite-builders that have made the last year unforgettable, you know who you are!
Jen bela okazo por saluti amikojn kaj amikinojn en la tuta mondo!

Tenslotte wil ik mijn ouders bedanken, die me altijd vrij gelaten hebben om mijn eigen weg te kiezen—ze zouden zo trots zijn geweest. Als afsluiting: Sandra, Esther en Ede, dank voor jullie geduld gedurende het hele traject, en voor jullie rol in mijn leven, die me heeft doen realiseren wat ècht belangrijk voor me is.

Abstract

The overall goal of the work described in this thesis is: "To design a flexible tool for state-of-the-art model-based derivation and automatic application of black-box tests for reactive systems, usable both for education and outside an academic context." From this goal, we derive functional and non-functional design requirements. The core of the thesis is a discussion of the design, in which we show how the functional requirements are fulfilled. In addition, we provide evidence to validate the non-functional requirements, in the form of case studies and responses to a tool user questionnaire.

We describe the overall architecture of our tool, and discuss three usage scenarios which are necessary to fulfill the functional requirements: random on-line testing, guided on-line testing, and off-line test derivation and execution. With on-line testing, test derivation and test execution takes place in an integrated manner: a next test step is only derived when it is necessary for execution. With random testing, during test derivation a random walk through the model is done. With guided testing, during test derivation additional (guidance) information is used, to guide the derivation through specific paths in the model. With off-line testing, test derivation and test execution take place as separate activities.

In our architecture we identify two major components: a test derivation engine, which synthesizes test primitives from a given model and from optional test guidance information, and a test execution engine, which contains the functionality to connect the test tool to the system under test. We refer to this latter functionality as the "adapter". In the description of the test derivation engine, we look at the same three usage scenarios, and we discuss support for visualization, and for dealing with divergence in the model. In the description of the test execution engine, we discuss three example adapter instances, and then generalise this to a general adapter design. We conclude with a description of extensions to deal with symbolic treatment of data and time.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Chapter 1

Introduction

The overall goal of the work described in this thesis can be summarised as follows: "To design a flexible tool for state-of-the-art model-based derivation and automatic application of black-box tests for reactive systems, usable both for education and outside an academic context." We refer to our test tool design as TorX[1]. The design that we describe in this thesis encompasses work on several test tool implementations, in particular TorX [BFdV+99, TB03a] and JTorX [Bel10]. TorX was our first implementation, in which we shaped our design. It was developed for a large part during a research project called Côte de Resyste [TB03b]. JTorX, developed several years later, is a reimplementation of TorX, created with ease of deployment in mind. We give more information about TorX and JTorX in Appendix A.

In this introduction, we provide a bridge between the high-level summary above and the remainder of this thesis. We start by introducing the concepts mentioned in the above summary. We then discuss the functional and non-functional requirements that we "impose" on the tool, and discuss how we validate them. For the functional requirements, this validation consists of indicating where, in the remainder of this thesis, they are introduced into the design of the tool. For the non-functional requirements, we discuss how to show that our tool satisfies them. We end this chapter with an outline of the remainder of this thesis.

1.1 Concepts

We introduce the concepts of the summary above one by one, where we first look at concepts that have to do with the functionality of the tool. We start with a discussion of "reactive systems", followed by a general introduction to testing, and a discussion of various kinds of testing, of which "black box" is one. We then discuss testing activities, where we explain "test derivation" and "test application"; we discuss test automation; and we give an introduction to

[1] The acronym TorX– Testing Open arCHitecture – was chosen during the Côte de Resyste project [TB03b] in which the tool TorX was originally developed.

"model-based" testing, and explain what we mean by "state-of-the-art" in that context.

Reactive systems The systems that we consider for testing are so-called *reactive systems*. For this kind of systems, the system behaviour is largely driven by the behaviour of the environment, i.e. the system behaviour is a reaction to the behaviour of the environment. Reactive systems are (almost by definition) open systems: systems that depend on an environment (which would be a user or another part of a larger system) that interacts with them. In contrast, a closed system, like a system that prepares a batch of monthly salary statements from an employer's employee database, is typically not reactive—such a system just transforms input data into output data. Many (most?) systems that we use everyday in daily life are reactive. A typical example of a reactive system is the plain old telephone system. We pick up the handset of a phone and we hear a dialling tone. We start dialling and the dialling tone stops. Once we completed dialling a number we hear an indication of the dial ed phone ringing, or tones that tell us that the dial ed number is busy or is not in use. This example nicely shows how a reactive system reacts to the actions of the user (picking up the handset, dialling a number) and how the user can observe the reaction of the system to these actions (the various tones heard, and the silence). For testing the behaviour of such system—the kind of testing that we focus on in this thesis—we can build on this interaction: We provide stimuli to the system (we pick up the handset) we observe what the system does (we listen whether we hear a dialling tone), and then we compare the observed system behaviour (dialling tone or silence or something else) to the expected behaviour (dialling tone).

Testing *Testing* is the activity of assessing some quality of a system—our focus is on *reactive* systems—by means of experimentation, i.e. stimuli are applied to the system, and its responses to these stimuli are observed, and evaluated, which typically leads to a so-called verdict (discussed below). The system that we want to test is called the *Implementation Under Test* (abbreviated as IUT). In Figure 1.1 we show testing in a very abstract way.

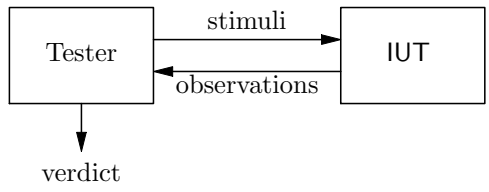


Figure 1.1: Tester with Implementation Under Test and Verdict

Testing is an important means to ascertain the quality of systems, by attempting to find errors in them (and thus allowing these to be repaired), as well as to increase confidence in the quality of systems, by showing that important functionality works as expected. As the *V-model* [Roo86] shows, during

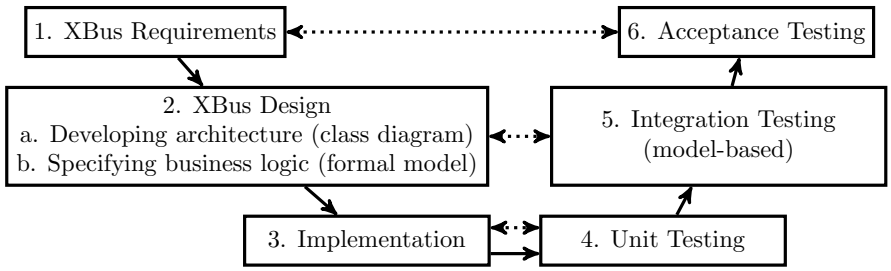


Figure 1.2: The V-model used for development of XBus, see Chapter 8.

the construction of systems, considerable effort is spent on testing on many levels—from the unit test of small parts via integration tests when components are combined till acceptance tests on the entire system when it is handed over to a client. When parts of a system are revised, regression tests help to ensure that functionality untouched by the revision continues working as expected. In Fig. 1.2 we show the V-model used in the development of a software bus called the XBus; we discuss development and testing of the XBus in Chapter 8.

Of course, testing is not the sole activity that is decisive when it comes to the quality of systems. Testing points out errors in the implementation of a system, but it does not build the system. And though it may help to improve the confidence in the quality of systems, testing can only demonstrate the presence of errors, not guarantee their absence. During the construction of a system, other activities help to improve its quality. Validation of the design, for example using techniques like model-checking, helps to catch design errors before the system is implemented. During implementation traditional techniques, like code reviewing of central parts of the code, remain useful. New techniques like model-checking the implementation of a system itself (instead of only a model of it), may in a nearby future be an important complement to testing.

All techniques available to control and improve quality of systems, of which testing is just a single one, have their particular strong points. Using the techniques in combination is the best way to build better systems, in particular because that avoids using testing as the 'catchall' for all errors. Moreover, errors that are found soon after they have been made are relatively cheap to repair; a design error that is found only when the complete system is tested may be very expensive to repair. That said, testing right now remains one of the chief techniques to catch errors before systems are put into production use.

Kinds of testing There are many different kinds of testing. In the first place, testing differs based on the aspect one wants to assess. For example, usability testing tries to discover how easy it is to use a system; robustness testing tests how well the system deals with other inputs than those anticipated for 'normal' use of the system; stress testing tests how well the system can cope with a high load; interoperability testing tests whether two or more systems can work together (communicate with each other); *conformance testing* tests whether a system complies with a functional specification of it. In this thesis we focus on

conformance testing.

As we have seen above, testing also differs in the system level on which it is applied: whether on individual units, modules, combinations of modules, subsystems or complete systems.

Testing can also differ in who is doing the testing (or for whom the testing is being done). For example, a software module can be tested by a developer who tests his own work, but also by an integrator who tests the modules he has to combine; a system can be tested by a user before accepting it, but also by an independent testing institute that tests it, e.g. for certification.

Another difference is the distinction between *black box* and *white box testing*. When we cannot look inside the system and do not not know its internal structure but can only use the interfaces that it offers to interact with it, we say that we are *black box testing*. This is the natural way of testing when doing conformance testing, because then we test whether the system behaviour conforms to a given specification in terms of its interactions with its environment. When we can look inside the system and do know its internal structure, and can use that for testing we are *white box testing*. Between black and white there usually is grey; also here. When we have a limited view on the internal structure of the system that we test we are *grey box testing*.

The last difference that we mention is about the relation between two testing activities: *test derivation* (making, designing, or in any other way obtaining the test "experiments", which are typically referred to as *test cases*) and *test execution* (applying the test cases, i.e. performing the test "experiments" on the system under test). (Below we discuss these activities in more detail.) Test derivation and test execution are usually done in separate phases, where the *test suite* (the collection of test cases) is the intermediate result. We call that approach *off-line testing*, it is also referred to as *batch testing*. We illustrate this approach in Fig. 1.3, which we obtained from Fig. 1.1 by decomposing component "Tester" into the combination of "Test Derivation", "Test Execution" and "Test Suite". A different approach is to derive a test on demand during execution, which for example may occur when a programmer is exploring freshly written code and uses observations made so far as inspiration for the next stimulus to give. In that case there is no test suite as intermediate result. We call this approach *on-line testing*, it is also referred to as *on-the-fly testing*.

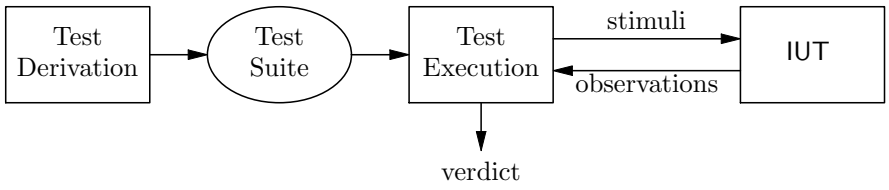


Figure 1.3: Off-line Testing, with Implementation Under Test and Verdict. Note how component "Tester" of Fig. 1.1 has been decomposed into the combination of "Test Derivation", "Test Execution" and "Test Suite".

Conformance testing As seen above, conformance testing is the activity of testing whether or not a system implementation conforms to (is a valid implementation of) a functional specification of it. The functional specification of a system prescribes the behaviour of the system—how it should interact with its environment, which can be seen as the user (in a broad sense) of the system. The functional specification prescribes nothing about the implementation of the system, not even about its structure. In a way, it only defines the interface between the system and its user. In an ideal world such specification could be the basis for the implementation of a system, in which case the activity of conformance testing allows testing whether the implementation behaves as expected.

Conformance testing originated in the application domain of communication protocols. Typical examples of such protocols are those that 'run' the Internet like IP [Int81], UDP [Pos80], TCP [Pos81], and, to give an example of an older system, those that make the 'plain old telephone system' work. Such protocols need a rigid and unambiguous specification to allow multiple parties to independently create implementations that can cooperate (interoperate). For this purpose such protocols are typically standardised. The existence of a standard almost automatically raises the question whether a particular implementation complies with the standard—a question to which conformance testing can provide an answer.

The answer that conformance testing provides is a judgement about the correctness of a system: a *verdict*. Verdicts are issued on multiple levels: they are associated with the result of an individual test run, and also with the result of an entire test suite. Typical verdicts are *pass*, *fail*, *inconclusive* and *error*. Verdicts *pass* and *fail* are used to indicate whether or not the system complies with the specification, at least as far as the tested behaviour is concerned. It may not be possible to give such conclusive verdict. Verdict *inconclusive* is used when a system does not exhibit the behaviour that we want to test, but at the same time does not produce an error. This may be the case with a non-deterministic system that at a certain 'point' can choose between multiple behaviours, and then chooses a different behaviour than the one that we want to test. Verdict *error* is used when an error occurred during testing, not because of errors in the implementation under test, but because of errors in the tester.

Testing activities Usually the activity of testing is subdivided into activities *test derivation* and *test execution*, already mentioned in the discussion of kinds of testing, above. When testing is used to find errors in order to repair them, a third important activity is *analysis* of the test execution results (if testing is only used to decide whether the system is correct or not, one may not care about the details of the errors).

The activity of test derivation is responsible for deciding which stimuli to provide, in which order, and for deciding which observations will be interpreted as indicative for correct behaviour. Each *test case* that results from this activity consists of *test steps* which represent the interactions (providing stimuli, making observations) with the system. A test case may have an associated *test purpose*: the objective of the test case.

The activity of test execution is responsible for doing the actual experimentation with the system that is to be tested, i.e. to provide stimuli and make observations, and to evaluate whether the observations obtained can be interpreted as indicative of correct behaviour.

For test result analysis the most important ingredients are the sequences of stimuli and observations that result from execution. There may be more information that can be used, information produced by the implementation that is not part of its "normal" interaction with its user, and thus not part of the observations used in testing. Examples could be diagnostics or debugging messages. Essentially all information that can help to get insight in what the implementation was doing when it was in error may be helpful.

Test selection and scheduling In general, it will be neither possible nor desirable to execute all imaginable (or derivable) test cases on a system: selection is necessary. In addition, in some cases the order in which test cases are executed matters, and then scheduling of the selected test cases is necessary too. Selection can be done in each of the activities of derivation and execution: one can be selective about which tests to derive, and one can be selective about which tests to execute. We mention three ways in which one can do test selection, and then we discuss scheduling.

Firstly, one can use random selection: whenever, in the derivation or execution activity, there is a choice among multiple alternatives, one makes a random choice. One can use random selection as the complete and sole selection strategy, but also as "fallback" strategy in combination with other selection strategies. In the latter case, when other selection strategies are not fully decisive, but leave a choice among multiple alternatives, random selection is used to resolve such choice.

Secondly, in addition to the information about the behaviour of the system, there may be other information about the system that can be used for selection, like: (1) behaviour that the user wants to observe (or to not observe), i.e. a test purpose, or (2) information about typical interaction scenarios, or (3) information about the (relative) importance of (testing) certain behaviours.

Finally, there may be information from previous test runs, (or even from the test that is actively being derived) that can be used in the test selection process, for example: coverage information. Coverage information is about how much of the requirements, or how much of the model from which the tests are derived, or how much of the implementation (code), has already been covered by the tests derived or executed so far. Using such coverage information, one then derives or selects the test or tests that will increase the coverage most. In the case of on-line testing, one can use information obtained from the test steps executed so far, during the derivation of additional test steps for the current test run—one can imagine tool-guided exploratory testing, where choices in the test derivation process are resolved by a user that interacts with the testing tool.

When multiple test cases have to be executed, one has to choose in which order the individual tests are executed (unless in the test execution activity all tests are executed in parallel). Although in many cases the order may not matter—in each run, all test cases are always executed—there may be cases

where such order is important. For example, when risks or costs are associated with errors, one may want to start with those test cases that expose errors with the highest risk resp. the highest cost. Other criteria to schedule the order in which test cases are executed, may be, for example, the importance of functionality to the user, the cost or availability of the resources necessary for the execution of each of the test cases, the time that each of the test cases takes to execute, or the dependencies between the test cases. Dependencies between the test cases can be such that if one test fails other tests become irrelevant because the features they test for depend on functionality that has been shown to contain errors already.

Test automation Testing can be a laborious task with lots of repetition, which makes it error-prone, and at the same time an ideal candidate for automation, for both the activities of test derivation and test execution.

For test execution, there are many solutions that help to automate it. The so-called 'capture and playback' tools essentially repeat (playback) on command (pieces of) a task that has been demonstrated (captured) once. Other tools read a test case or test suite in some given format and automatically execute each of the test steps. Such tool will typically either be only usable for a certain class of systems, or it will contain some kind of "glue code" that can be adapted to match the (kind of) system it has to interact with. There are tools that help to realise the actual interaction with the system under test. For unit testing, libraries exist that allow one to easily define (program) tests for the software modules one is working on, and to execute these. Also support for regression testing exists.

For test derivation, automation is less common: in many cases, test derivation is done manually, from ad-hoc quick testing by a programmer to 'test if it works' to systematic manual derivation of tests from the system requirements or system specification. This does not mean that automatic derivation of tests is not done at all; to us, the most promising way do automatic test derivation is by using model-based testing, which we discuss below.

Model-based testing In model-based testing, tests are derived algorithmically from a formal *model*, typically in an automated way. This means that the model must be available in a form that allows automatic processing, and that the model must have a precise meaning (semantics). To us, there is a natural match between conformance testing, where we want to check for conformance of a system to a specification, and model-based testing, where we automatically derive tests from a given model. This match is realised (made effective) by deriving the tests, necessary to check conformance, automatically from a model that describes the behaviour of the system in terms of its interactions with its environment.

Of course, such automatically derived tests must not give *fail* verdicts for implementations that we (intuitively) would consider conforming, i.e. they have to be *sound*. Moreover, we want to have some form of guarantee that the automatically derived tests do not consistently overlook certain errors, i.e. the derivation algorithm must have no inherent "blind spots"; it must be *exhaustive*.

The formal framework for conformance testing, which we discuss in Chapter 2, allows us to reason about this.

Test architecture In Figure 1.1 we depicted that the tester can directly interact with the IUT. In practice this may not always be the case, for example because the system that we want to test (i.e., the IUT) is actually a subsystem of a bigger system from which it cannot be isolated for testing. Then, we have to interact with the IUT 'through' this bigger system. A *test architecture* gives an abstract view of how the tester interacts with the IUT. It contains the IUT, the tester, the *test context* and the interfaces between them. Figure 1.4 shows an example.

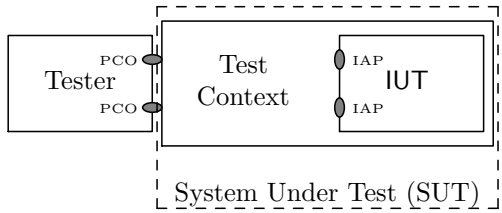


Figure 1.4: Test Architecture with test tool, SUT, test context and IUT.

The *test context* consists of those parts of a system that are not the object of testing, but nevertheless are present when the IUT is tested. For testing we usually assume that these parts are correct. From now on we say that the tester interacts with the *system under test* (abbreviated as SUT), i.e. with the combination of IUT and test context. The interfaces that the tester uses to interact with the SUT are called the *points of control and observation* (PCOs). The interfaces by which the IUT offers access to its environment (its user, in a broad sense) are called the *implementation access points* (IAPs). Ideally, the test context is empty, such that the SUT coincides with the IUT, and the IAPs coincide with the PCOs, but often this is not the case. We are not very strict, though: when the difference between IUT and SUT does not matter for the discussion at hand, we may use either SUT, IUT or "the implementation" to refer to the system under test.

Model-driven testing? Beyond the notion "model-based" there is, in our view, the notion "model-driven". To our understanding, the model-driven development approach takes automation a step further than we do with model-based testing. In the model-driven development approach the focus is on automation, using (domain) models as input—all other artefacts are either generic or automatically created from the models.

In our model-based testing approach we do have artefacts that are automatically created from models (test cases), and generic artefacts (the test tool), but also at least one artefact that is, at least in some cases, neither generic, nor automatically created: the "glue code" that allows the (generic) testing tool to interact with the SUT. Typically, when, for a given SUT, there is no generic "'glue code" available, we create ad hoc "glue code", by hand.

A natural extension of our "model-based" approach would be to use, in addition to models that describe the behaviour of the SUT, also models that describe how the SUT interacts with its environment, such that also the "glue code" can be automatically derived.

In this thesis we do discuss the functionality that has to be provided by the "'glue code", but we leave the ability to automatically generate the "glue code" from a model for future research. In this sense, we limit ourselves to model-based testing.

1.2 Design Choices and Design Goal

Now that we discussed the concepts used to formulate the overall design goal at the start of this chapter, we discuss how we mapped this overall goal on functional and non-functional requirements. We do this in two steps. First, in Section 1.2.1, we discuss the design choices that we made, i.e. those choices that we made at the start of the design, to focus the design. These choices constrain the design space. Then, in Section 1.2.2 we discuss the design goal, i.e. we present the functional and non-functional requirements.

1.2.1 Design Choices and Criteria

State-of-the-art theory An important decision was to use existing, state of the art theory, and not work on theoretical extensions. The theory that we chose consists of a formal framework for conformance testing (we give an overview in the first part of Chapter 2, 'Theoretical Foundation' on page 21)—together with an instantiation of that framework (presented in the second part of Chapter 2). The framework formalises the concepts that play a role in conformance testing, like model, implementation, test, observation, verdict, execution, but also the concept of conformance itself—what does it take for a system to conform to a specification—as abstract concepts, for which a concrete instantiation can be chosen. We chose to instantiate the framework with Tretmans' **ioco** theory [Tre96, Tre08], with models that are (can be interpreted as) labelled transition systems.

Modelling formalism and language The **ioco** theory that we use is defined on models that are labelled transition systems (LTSes), i.e. models that describe the behaviour of a system in terms of states and transitions between the states, where the transitions are annotated with (action) labels that represent actions of the system (interactions between the system and its environment). Thus, we chose LTSes as the (abstract) model type on which we base our design.

For very small systems (toy examples) we may write (or draw) an LTS by hand, but for bigger systems, describing the behaviour directly as an LTS is cumbersome and, worse, error-prone. To describe the behaviour of bigger systems we typically use more higher-level description languages, like the process-algebras LOTOS [ISO89], mCRL2 [GKM$^+$08], or Promela [Hol91], and then derive the LTS from such higher-level description. Because, on the one hand,

we had no clear preference for a higher-level modelling language, and on the other hand, we wanted to be able to reuse existing tool support for such languages, we decided that for our tool design we should strive for independence of modelling languages.

Coping with large models When describing a system in a higher-level language, it is not uncommon to obtain a model of which the LTS has a high, or even infinite, number of states and transitions (state space). First creating such large (or infinite) LTS from a model, and then reading the LTS into the testing tool, may take a relatively long time (or not finish at all, in the case of an infinite state space).

The **ioco** theory can handle models that have a large, even infinite, number of states, as long as each state has a finite number of outgoing transitions (is *finitely branching*), and as long as there is no infinite sequence of internal transitions (cycles of internal transitions are o.k.). The **ioco** theory can handle large models, because in the test derivation algorithm, for each test step, the algorithm only looks at the outgoing transitions of the "current set of model states". This set of model states initially only contains the initial state of the model; after a test step this state set is updated to contain the destination states of those transitions that correspond with the current test step.

Thus, for test derivation with **ioco** we can cope with large models by obtaining the LTS from the model on demand (on-the-fly), during (concurrent with the) test derivation, as needed by the activity of test derivation. We chose to use this way of accessing the LTS for our design, for two reasons. Firstly, because, as we discussed already, it allows us to deal with large models. Secondly, because it seemed the most general way of obtaining the information necessary for test derivation from a higher-level language model—and such generality seemed beneficial to our design, given our decision to strive for independence of the modelling language. (A side effect of this choice is that we may not know how many states and transitions the LTS of our higher-level model contains, which may affect e.g. the ability to compute model coverage criteria like state-transition coverage).

Black-box testing An immediate consequence of the choice for the **ioco** theory is the choice for black-box testing, with models that describe the interaction between the system under test and its environment: that is what the **ioco** theory is about. And, as we wrote when we mentioned black-box testing as one of the "kinds of testing" in Section 1.1: this is the natural way of testing, when doing conformance testing. An advantage of this choice for black-box testing is that we have rather limited requirements on the implementations that we will test: We only have to be able to interact with them, i.e. it will not be a problem when we do not have access to the actual implementation, for example because it is at the other side of the network connection via which we access it, or because it is running on (inside) an embedded device.

Test selection and test execution The theory describes how to derive test cases, but does not say anything about test selection, nor about the relation

between test derivation and test execution (i.e. about the choice between on-line and off-line testing), allowing us to take our own decision in this regard. We decided that our design should be suitable for both on-line and off-line testing. That said, we chose to focus mostly on on-line testing, because it is easier to implement. Regarding test selection, we chose to support two approaches that were based on available theory. Both approaches affect the test derivation activity, because of our focus on on-line testing.

1. Our first selection approach is a simple one: to use a *random walk* in the (state space of the) model.
2. Our second selection approach is based on the use of *test purposes*, where, to us, a test purpose is a model of a part of the system behaviour that we want to focus on during testing. During test derivation, a given test purpose is used to guide the test to a part of the the system behaviour (and thus, the corresponding part of the state space) that we want to focus on.

1.2.2 Design Goal

The overall design goal stated at the beginning of this chapter can now, with the help of the concepts and design choices discussed above, be broken down into functional requirements (denoted by (f)) and non-functional requirements (denoted by (nf)).

Functional requirements Expressed as functional requirements, our design goal is:

1 (f). the tool should be based on **ioco** theory;
2 (f). the tool should work on models that have an LTS semantics;
3 (f). the tool design should be suitable for both on-line and off-line testing;
4 (f). the tool should support on-line testing;
5 (f). the tool design should be independent from particular modelling languages;
6 (f). the tool should support very large and infinite state space models;
7 (f). for on-line testing, the tool should support random mode and guided mode;
8 (f). the tool design should make no assumptions about the SUT, except that it is a reactive system.

Non-functional requirements In the design goal summary we mentioned the following high-level non-functional requirements: (a) the tool should be flexible, (b) it should be usable for education, and (c) it should be usable outside an academic environment. These high-level requirements entail more concrete requirements, some of which are functional, and some of which are non-functional.

Ad a: Flexibility Flexibility of the tool translates to the following requirements about evolving the tool.

9 (nf). it should be easy to accommodate theoretical progress;
10 (nf). it should be easy to incorporate new conformance relations;
11 (nf). it should be easy to incorporate new test selection strategies.

Ad b: Usability for education Use in education involves the following two scenarios. Firstly, the tool should be usable in courses at bachelor or master level, to allow the students to experience the concept of model-based testing. In this scenario, the students use the tool not only to test real (toy) implementations, but also to compare models by testing (i.e. to obtain an answer to the question: does an implementation, represented by given model i, conform to the specification, represented by model m). Secondly, the tool should be usable to explain, on an intuitive level, the basic principles of model-based testing, to almost any person (but in particular: testers and managers). This means:

12 (nf). it should be easy to deploy the tool (install and use);
13 (f). it should be easy to create a simple model (like an automaton) for use with the tool;
14 (f). the tool should provide insight in the theory and algorithms that it implements, e.g. by visualisation;
15 (f). it should be possible to use a simulated model as system under test;
16 (nf). it should be simple to connect the tool to toy implementations (this requirement is subsumed by requirement 24 below).

Ad c: Usability outside an academic context Usability outside an academic context involves two scenarios. Firstly, the tool should be usable by students that need model-based testing functionality when doing an internship or an external graduation project. Secondly, the tool should be usable by the people (like experienced testers) at the company where the internship or graduation project takes place, or where a case study (that involves model-based testing) takes place. This means:

12 (nf). (as for educational use) it should be easy to deploy the tool (install and use);
17 (nf). it should be possible to use the tool without being an expert in the theory that the tool implements;
18 (f). the design should allow use of modelling languages suitable for non-experts;
19 (f). the design should allow use of modelling languages with suitable expressive power;
20 (f). it must be possible to validate the models, either in the tool, or using external tools;
21 (f). the tool should produce/keep test execution data for analysis;
22 (f). the tool should be correct (i.e. it should correctly implement the theory);
23 (nf). the tool should have sufficient performance to be usable;
24 (nf). it should be easy to connect the tool to the system under test (this requirement subsumes requirement 16).

1.2.3 Validation

Here we discuss how the design goal can be validated. In Chapter 8 we do the actual validation. In that chapter, we evaluate to what extent the design goal is fulfilled. That evaluation is preceded by a discussion of the "evidence" that is used in the evaluation: two implementations of our design, TorX [BFdV+99] and JTorX [Bel10], case studies done with these implementations, their use in research and industry, and results of a questionnaire posed to users.

Below, we discuss each of the requirements in turn.

Ad 1 (f): the tool should be based on ioco theory. In Chapter 2 we show the **ioco** test derivation algorithm. In Chapter 3 we discuss the architecture of our tool for model-based testing, and we show how the **ioco** test derivation can be performed by the components in our architecture.

Ad 2 (f): the tool should work on models that have an LTS semantics. In Chapter 3 we discuss our architecture for test derivation and execution. In Section 4.2.1 we discuss the decomposition of the test derivation component of that architecture into two sub-components: (1) a modelling-language-specific one, that provides access to the LTS of the model, and (2) a generic one, that does the **ioco** test derivation using the interface provided by the modelling-language-specific sub-component.

Ad 3 (f): the tool design should be suitable for both on-line and off-line testing. In Chapter 3 we discuss our architecture for test derivation and execution. In Section 3.4 we discuss how it supports on-line testing; in Section 3.6 we discuss how it supports off-line testing.

Ad 4 (f): the tool should support on-line testing. In Section 3.4 we decompose the tool into components and show an algorithm for on-line testing that does its work using these components.

Ad 5 (f): the tool design should be independent from particular modelling languages. In Section 4.2.1 we show how we decomposed the test derivation component into two sub-components: (1) a modelling-language-specific one, that provides access to the LTS of the model, and (2) a generic one, that does the **ioco** test derivation using the interface provided by the modelling-language-specific sub-component. In this way, instances of the first sub-component hide modelling language specific details from the rest of the tool.

Ad 6 (f): the tool should support very large and infinite state space models. In Section 4.2.2 we discuss the interface between the modelling-language-specific and the modelling-language-independent test derivation sub-components; this interface offers on-the-fly model access, which provides support for infinite state space models.

Ad 7 (f): for on-line testing, the tool should support random mode and guided mode. In Section 3.4 we discuss the random mode, and in Section 3.5 the guided mode.

Ad 8 (f): the tool design should make no assumptions about the SUT, except that it is a reactive system. In our architecture, the adapter component provides an abstract interface to the SUT (i.e. the "glue code" that provides the connection between tester and SUT). This interface, which we discuss in Section 3.4.2, only assumes that we can provide stimuli to the SUT, and obtain responses from the SUT.

Ad 9 (nf): it should be easy to accommodate theoretical progress. In Section 8.5.2 we validate this requirement by discussing the impact on the design of our tool of (i) a change in the **ioco** theory, and (ii) the addition of the ability to deal with divergence (τ-loops).

Ad i: In the initial definition in [Tre96] test cases were not input-enabled, i.e. in a test step, in which the tester would try to apply a stimulus, the tester would not be willing to accept a response from the SUT. In [Tre08] this was revised to make test cases input-enabled: the tester is always willing to accept a response from the SUT.

We evolved the design of our tool over time, and, a number of years after our initial implementation TorX, we made a reimplementation JTorX [Bel10]. Initially, in TorX, we used the initial **ioco** definition; later, in JTorX, we used the revised definition.

Ad ii: The **ioco** theory of [Tre96, Tre08] explicitly rules out models with τ-cycles. In [Sto12, STS13] **ioco** is also defined in case of divergence. In Section 4.1 we discuss the changes necessary to our design to include basic support for divergence.

Ad 10 (nf): it should be easy to incorporate new conformance relations. In Section 8.5.2 we validate this requirement by discussing the impact of the addition of support for the **uioco** conformance relation [vdBRT04]. Initially, we only supported test derivation based on **ioco**; later we added support for **uioco**. (It should be noted, though, that **uioco** is very similar to **ioco**.)

Ad 11 (nf): it should be easy to incorporate new test selection strategies. In Section 8.5.2 we validate this requirement by discussing how we, for a case study, delegated the selection of the next test step to a separate component in the architecture.

Ad 12 (nf): it should be easy to deploy the tool (install and use). In Chapter 8 we discuss experience with the tool; this includes experience reported by users in publications about case studies done with the tool, and experience reported in response to a questionnaire. In Section 8.5.3 we use this experience to validate this requirement.

Ad 13 (f): it should be easy to create a simple model (like an automaton) for use with the tool. As indicated above, the tool design allows the use of multiple different concrete modelling languages. JTorX has built-in support for the *GraphML* [Gra12] format that is written by the graph editor yEd [yWo]. This graph editor allows creation of models by drawing them. Moreover, the layout of *GraphML* models is respected by the visualisation of JTorX. We discuss the former in Section 4.2.4 and the latter in Section 4.2.5. We use a questionnaire to get information about the user's perception.

Ad 14 (f): the tool should provide insight in the theory and algorithms that it implements, e.g. by visualisation. We have chosen to provide insight in the test derivation algorithm, by using visualisation to show the relation between the model and the test steps that are derived from it. In Section 4.2.5 we discuss the impact of this choice on the design. Moreover, JTorX contains an interactive simulator (see Section A.2), which is coupled to the visualisation functionality. We do not discuss details of the interactive simulator in this thesis.

Ad 15 (f): it should be possible to use a simulated model as system under test. As we discuss in Chapter 5, in our design the adapter component provides the connection (the "glue code") between the generic functionality of our tool, and a given SUT. We have a few generic instances of the adapter component, one which allows us to use a simulated model as SUT. We mention this adapter instance in the introduction to Chapter 5.

Ad 16 (nf): it should be simple to connect the tool to toy implementations. Among the generic instances of the adapter component which are mentioned above Ad 15 (f), there are two that are specifically designed to make it simple to connect to a toy SUT. These adapter instances expect that the SUT interacts either on its standard input and standard output, or via a TCP connection. They also expect that the SUT interacts using messages that are a textual representation of the interaction description in the model, i.e. messages that are a textual representation of the labels of the LTS. We discuss these adapter instances in Section 5.1.1.

Ad 17 (f): it should be possible to use the tool without being an expert in the theory that the tool implements. In Chapter 8 we discuss the experience with the tool. In Section 8.5.3 we discuss to what extent the reported experience allows us to validate this requirement.

Ad 18 (f): the design should allow use of modelling languages suitable for non-experts. Any language, of which the semantics can be expressed as an LTS, can be used with the tool, as long as a modelling-language-specific model-access component for the language is available. It may not always be feasible to construct such model-access component. However, if there is an

interactive simulator for the language available, and an API to access the LTS-related data structures inside it (states, transitions, labels), it may be possible to construct a model-access component, based on the simulator. In Section 4.2.2 we discuss the interface that a model-access component must provide.

Ad 19 (f): the design should allow use of modelling languages with suitable expressive power. In Chapter 8 we discuss the experience with the tool. In particular, we list the case studies that have been carried out, and with each case study we indicate the modelling language (formalism) that was used. In Section 8.5.3 we use this information—where we pay special attention to the suitability of the modelling language to represent the behaviour of the SUT—to validate this requirement.

Ad 20 (f): it must be possible to validate the models, either in the tool, or using external tools. For modelling formalisms that have their own tool environment, typically this tool environment contains functionality to validate the model, like an interactive simulator, or model-checking functionality. However, not all modelling formalisms that we support in JTorX have a tool environment that contains this kind of functionality. Examples of such formalisms are the *GraphML* formalism, and the GraphViz input language. As discussed in Section A.2, we integrated an interactive simulator in our tool (in the JTorX implementation of our design), to provide at least some validation functionality for such formalisms.

Ad 21 (f): the tool should produce/keep test execution data for analysis. The tools produce a test log that contains, for all test steps that were executed, (i) a timestamp, (ii) the model representation of the interaction with the SUT, (iii) information about the states and transitions that have been traversed in the model, (iv) a concrete representation of the interaction with the SUT, when reported by the "glue code" that connects the tool to the SUT. We don't discuss further details about these logs in this thesis. In Chapter 7 we discuss a case study in greater detail. There we present information that is obtained from log file analysis during the case study.

Ad 22 (f): the tool should be correct. What we mean is: the tool should not give a *fail* verdict for a correct (conforming) implementation (i.e. the tool should be *sound*), and, moreover, testing a non-conforming implementation with the tool should eventually—ideally in a reasonable amount of time, even though in practice it may take longer than we are willing to wait—produce a *fail* verdict (the tool should not have inherent blind spots). In Chapter 8 we discuss experience with the tool; in none of the cases correctness of the tool was an issue. In one of the case studies, discussed in Section B.1, we created 25 mutants from an assumed-to-be-correct program, by introducing errors into it, and used the tool to identify the erroneous mutants. To our surprise, the tool was unable to find errors in two of the mutants. After analysis, we found that

the model, used for testing, was incomplete in one aspect, and did not contain the scenario that was necessary to trigger that particular error.

Ad 23 (nf): the tool should have sufficient performance to be usable. In Chapters 7 and 8 we discuss the experience with the tool. In all cases tool performance sufficed to carry out the case study, even when in some of the case studies it initially took some work to obtain sufficient performance. In most of the discussed case studies we did not look at performance in detail. We did study performance of the tool in more detail in the case study discussed in Chapter 7. In Section 8.5.3 we use the experience results to validate this requirement.

Ad 24 (nf): it should be easy to connect the tool to the system under test. In Chapter 8 we discuss experience with the tool; for each case study we indicate how we connected the tool to the SUT. In Section 8.5.3 we discuss to what extent this experience allows us to validate this requirement.

1.3 Overview

For easy reference, we list the functional requirements in Table 1.1, and the non-functional ones in Tables 1.2 (non-functional requirements w.r.t. development of the tool) and 1.3 (non-functional requirements w.r.t. use of the tool).

	requirement
1	the tool should be based on **ioco** theory
2	the tool should work on models that have an LTS semantics
3	the tool design should be suitable for both on-line and off-line testing
4	the tool should support on-line testing
5	the tool design should be independent from particular modelling languages
6	the tool should support very large and infinite state space models
7	for on-line testing, the tool should support random mode and guided mode
8	the tool design should make no assumptions about the SUT, except that it is a reactive system
13	it should be easy to create a simple model (like an automaton) for use with the tool
14	the tool should provide insight in the theory and algorithms that it implements, e.g. by visualisation
15	it should be possible to use a simulated model as system under test
18	the design should allow use of modelling languages suitable for non-experts
19	the design should allow use of modelling languages with suitable expressive power
20	it must be possible to validate the models, either in the tool, or using external tools
21	the tool should produce/keep test execution data for analysis
22	the tool should be correct

Table 1.1: Functional requirements

	requirement
9	it should be easy to accommodate theoretical progress
10	it should be easy to incorporate new conformance relations
11	it should be easy to incorporate new test selection strategies

Table 1.2: Non-functional requirements w.r.t. development of the tool

	requirement
12	it should be easy to deploy the tool (install and use)
16	it should be simple to connect the tool to toy implementations
17	it should be possible to use the tool without being an expert in the theory that the tool implements
23	the tool should have sufficient performance to be usable
24	it should be easy to connect the tool to the system under test (subsumes 16)

Table 1.3: Non-functional requirements w.r.t. use of the tool

1.4 Synopsis

The rest of this thesis is organised as follows.

Chapter 2 serves as starting point for the remaining chapters in this thesis: it provides the reader with a formal background for the remaining chapters. This chapter only recapitulates theory published by others.

Chapter 3 introduces the global architecture of TorX. The main decomposition into components was, in my recollection, more or less a group decision in the Côte-de-Resyste project. Our contribution is in the design of the interfaces and the algorithms.

Chapter 4 discusses the test derivation engine of TorX. First the basic test derivation engine is introduced; subsequently it is shown how this can be extended with means to guide the test derivation. Again, the main decomposition into components was a group decisions; our contribution lies in the interfaces and the algorithms.

Chapter 5 discusses the test execution engine of TorX, and in particular the TorX component (to which we refer as Adapter) that is responsible for the connection to and interaction with the system under test. We first present three example Adapter instances, and then present our Adapter design. Our contribution consists of the design of the example Adapter instances (even though typically case studies are a collaborative effort, we did design these), and of our Adapter design.

Chapter 6 discusses extensions to deal with symbolic treatment of data and time. Our contribution to this chapter is the (A)PTS formalism, which we use to describe the interfaces in the symbolic setting, and the extensions to our design that allow us to use the symbolic and timed Explorer components, which

were all (with the exception of SmileExp) designed and developed by others. The other theory presented in this chapter (e.g. about STS) recapitulates work published by others.

Chapter 7 presents a case study: Model-based specification, implementation and testing of a software bus at Neopost. The work in this case study was done in two phases: a first phase, which took place during the internship of a student at Neopost, and a second phase in which we tried to evaluate the thoroughness of the model-based testing that had taken place in the first phase. Our contribution consists of the work done in the second phase; in the first phase, the work was done by the student.

Chapter 8 presents an evaluation of our design w.r.t. the design requirements. This evaluation is preceded by the 'evidence' on which the evaluation is based, i.e., by a discussion of case studies that have been done with TorX and JTorX, of the use of TorX and JTorX for research and for education, and of the relevant responses to a questionnaire about them.

Chapter 9 summarises the thesis, presents overall conclusions and ideas for future work.

Appendix A discusses two implementations of our design: TorX and JTorX. Our contribution consists of the design and implementation of these two tools.

Appendix B gives additional details about the case studies presented in Chapter 8. Typically, case studies are an collaborative effort.

For the "Conference Protocol Entity" case, our contribution was on the test tool, the Adapter, and the actual test runs; the models and implementations were produced by other team members.

For the "EasyLink" case, our contribution was mostly on the test tool, and in the collaborative solving of problems as they arose once we started running the tests (like dealing with the unknown initial state of the television set); model and Adapter were created by other team members.

For the "Oosterschelde Storm Surge Barrier" case our contribution was in the model, in a small Adapter to connect TorX to the already existing test environment, and in running the tests.

For the "Myrianed Protocol Entity" case our contribution was again in creation of a model (this time for JTorX) and in a small Adapter to connect the testing tool to a testing environment created by the company that developed the Myrianed Protocol Entity, and in testing with JTorX. Other persons made the models, and ran the tests, with other testing tools.

For the "RiverCrossing Puzzle" case, our contribution was in creation of an Adapter, and of a visualisation of the IUT; the IUT itself was created by Mark Timmer (another member of the teaching team of the Testing Techniques course for which this "case" was developed), and the model was decided on by the team in a collaborative effort.

Appendix C gives the questions and responses of the questionnaire.

Chapter 2

Theoretical Foundation

In the previous chapter we formulated requirements on our testing tool design. In this chapter we give a brief overview of the main existing theory that forms the foundation of our work.

Model-based testing (black box conformance testing) studies the relation between a specification of a system and an implementation of that specification, i.e. the working system. The question that it tries to get an answer to is whether or not a given system implementation "is a correct implementation of" or "does conform to" a given specification. Because the implementation is treated as a black box, the method to get an answer is testing: performing experiments on the implementation and observing the responses. The left part of Figure 2.1 sketches the process of model-based testing. The idea is that to test whether implementation IUT "conforms to" specification s we apply test suite T_s. If IUT passes all tests, it conforms to s.

Mere conformance of an IUT to a specification s is no guarantee for a useful product, as we will try to illustrate with the following example. Imagine the following specification for a coffee- and tea machine. After a user inserted a coin of the right value, the machine must offer the user the ability to choose between coffee and tea, and then it will serve the chosen drink. However, the machine may also decide to immediately return the coin, instead of offering the choice between coffee and tea. This would be correct behaviour when the machine is unable to produce coffee or tea, for example because it has run out of coffee or tea. The choice between returning the coin and offering the choice between coffee and tea can not be controlled from outside of the machine. A machine that accepts a coin, and then immediately returns it would be a valid implementation of such a specification. However, to a typical user this would be a fairly useless implementation. Therefore, an extension to the process of model based testing has been proposed, orthogonal to the notion of "conforms to", that allows us to test whether an IUT "exhibits" interesting behaviour p—"interesting behaviour" p is referred to as a *test purpose*. The right part of Figure 2.1 visualises this extension. For the example above "interesting behaviour" would consist of the two scenarios in which the machine, after insertion of a coin, offers the user the choice between coffee and tea, and then produces the chosen drink.

2

In this chapter we present existing theory that allows formal reasoning about this process of model based testing. This theory is presented in four parts.

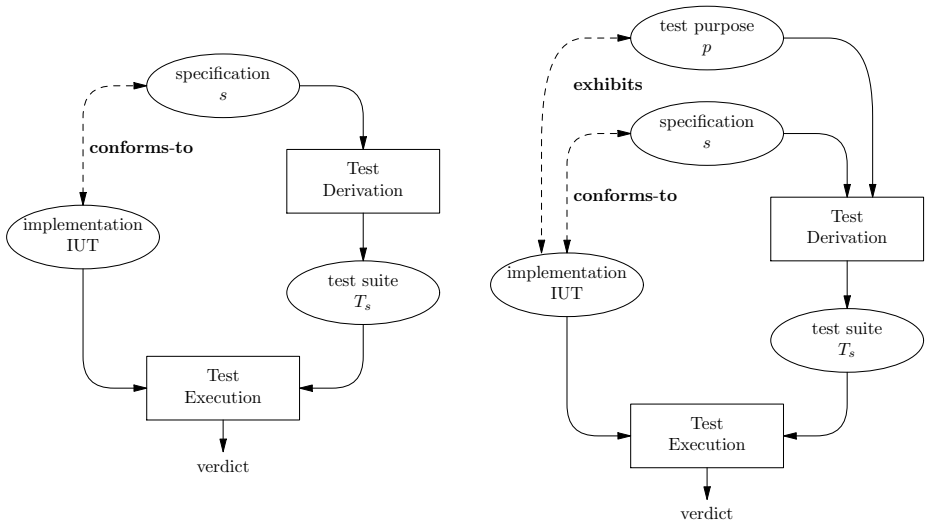


Figure 2.1: Left: Schematic view of model-based testing. Right: This framework, extended for testing whether the implementation exhibits a given interesting behaviour.

In Section 2.1, we show the general framework of *formal methods in conformance testing* which defines concepts and their relationships. A more detailed exposition of this framework can be found in [BAL+90, Tre94, ISO96, Tre99].

In Section 2.2 we will instantiate this framework with a particular notion of "conforms to". We have chosen to base our work on a conformance relation called **ioco** [Tre96, Tre08] (design requirement 1). Note that on a practical level the most important thing is that to be able to build tools we do need *a* notion of conformance; which particular notion of conformance is chosen may not be that relevant.

In Section 2.3, we show an extension of the general framework of formal methods in conformance testing with concepts that allow reasoning about when an IUT "exhibits" interesting behaviour.

In Section 2.4 we instantiate this extended framework with a particular notion of "exhibits".

Bibliographic note The remainder of this chapter is effectively a summary of chapters 2, 3, 4, and 5 of [Tre02] and of parts of [Tre08] (Sections 2.1 and 2.2), and of parts of [dV01] (Sections 2.3 and 2.4), mostly by omitting information deemed irrelevant for the presentation here, and paraphrasing or (almost) verbatim including relevant information.

This presentation differs with these sources in one respect: in this presentation we do not use the separate action θ to represent quiescence in test cases, but for simplicity we use δ instead.

2.1 Formal Framework for Conformance Testing

We reason about conformance testing in the formal world of mathematics; to do that we need in the formal world representations of the "ingredients" that we want to reason about, most of which are shown in Figure 2.1, i.e. the specification, the implementation, the conformance relation, the test suite, test derivation, test execution and the verdict. We assume *SPECS* to be the set of all specifications, and thus $s \in SPECS$; we assume *IMPS* to be the set of all implementations, so IUT $\in$ *IMPS*. Now we would like to have a formal relation between s and IUT, like **conforms-to** $\subseteq IMPS \times SPECS$, but that is where we face a problem: the system implementation is not a formal object, it is a concrete object like a piece of hardware or software and it is difficult to formally relate to that (we assume formal specifications written down in languages that have a formally defined semantics so specifications are formal objects). We work around this problem by making an assumption, the so-called *test hypothesis*, which says that each possible concrete implementation in *IMPS* has a corresponding model in the formal world. It is sufficient to assume that such formal model exists, we do not need to assume that we a priori know the model. We assume *MODS* to be the set of models of implementations, and we use $i_{\text{IUT}} \in MODS$ to denote the model of implementation IUT $\in$ *IMPS*. The test hypothesis allows formal reasoning about conformance between specifications and (models of) implementations, and formally expressing such conformance as a relation: the *implementation relation* **imp** (also referred to as *conformance relation*), where **imp** $\subseteq MODS \times SPECS$. We can now mirror the informal relation **conforms-to** between *IMPS* and *SPECS* in the informal world by the (formal) relation **imp** between *MODS* and *SPECS* in the formal world, as depicted in Figure 2.2, and we say that IUT is correct with respect to s, i.e. IUT **conforms-to** s, if and only if relation **imp** holds between i_{IUT} and s: i_{IUT} **imp** s.

$$\text{IUT } \textbf{conforms-to } s \ =_{\text{def}} \ i_{\text{IUT}} \ \textbf{imp} \ s \tag{2.1}$$

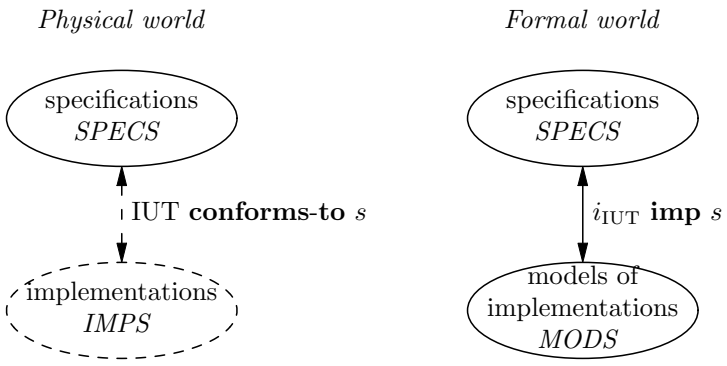


Figure 2.2: Relating implementations and implementation models with specifications.

Because the implementation is treated as a black box, we can only experience its behaviour by interacting with it: by performing experiments on it and observing the reactions. Also these experiments are formalised. The specification of each such experiment is called a *test case*, and the process of performing such experiment on an implementation under test is called *test execution*. A single test case may be applied to an implementation more than once; the application of a *test case* to an implementation is called a *test run*. Each test run results in an *observation*. We formalise this as follows. We assume *TESTS* to be the set of all test cases, and *OBS* the set of all observations. We represent the application of test case $t \in TESTS$ on implementation IUT (multiple times, when necessary to get all different test runs) by $\mathrm{EXEC}(t, \mathrm{IUT})$. Due to non-deterministic behaviour of the implementation different test runs of the same test case on the same implementation may result in different observations and thus we say that the result of $\mathrm{EXEC}(t, \mathrm{IUT})$ consists of a set of observations, i.e. it is a subset of *OBS*. Since IUT is not a formal object, EXEC is not a formal concept; it just represents the concrete execution of a test case. We mirror the concrete execution $\mathrm{EXEC}(t, \mathrm{IUT})$ in the formal world as $obs(t, i_{\mathrm{IUT}})$ using observation function $obs : TESTS \times MODS \to \mathcal{P}(OBS)$. Here we used $\mathcal{P}(OBS)$ to denote the powerset of *OBS* (the set of all subsets of *OBS*).

The essence of formal testing now is that we assume that we cannot distinguish the real IUT from its formal model i_{IUT} by executing tests and comparing observations (essentially formalising what is said in the *test hypothesis*):

$$\forall \mathrm{IUT} \in IMPS \; \exists i_{\mathrm{IUT}} \in MODS \; \forall t \in TESTS : \mathrm{EXEC}(t, \mathrm{IUT}) = obs(t, i_{\mathrm{IUT}}) \tag{2.2}$$

With the result of executing a test on an implementation a *verdict* is associated, which is either **pass** or **fail**. We formalise this using a family of *verdict functions* $v_t : \mathcal{P}(OBS) \to \{\mathbf{pass}, \mathbf{fail}\}$ that for each test case t associate a verdict with the set of observations that results from executing the test case. We can now define what it means to pass a test:

$$\mathrm{IUT}\ \mathbf{passes}\ t \ =_{\mathrm{def}}\ v_t(\mathrm{EXEC}(t, \mathrm{IUT})) = \mathbf{pass} \tag{2.3}$$

To pass a test suite $T \subseteq TESTS$ all tests in it must be passed:

$$\mathrm{IUT}\ \mathbf{passes}\ T \Leftrightarrow \forall t \in T : \mathrm{IUT}\ \mathbf{passes}\ t \tag{2.4}$$

The test suite is failed if it does not pass:

$$\mathrm{IUT}\ \mathbf{fails}\ T \Leftrightarrow \mathrm{IUT}\ \ \mathbf{pa\not{s}ses}\ \ T \tag{2.5}$$

We now have almost all the ingredients to talk about test execution in the informal world and to reason about conformance in the formal world. What is missing is how we get a test suite, and how we get the verdict functions for the test cases in it. We assume that for each implementation relation **imp** that we are interested in we can come up with a *test derivation* algorithm $gen_{\mathbf{imp}}$ that from a specification $s \in SPECS$ derives a set of test cases $T_s \subseteq TESTS$, i.e. we can see test derivation as a function $gen_{\mathbf{imp}} : SPECS \to \mathcal{P}(TESTS)$, with

the associated family of verdict functions. The actual algorithm and verdict functions depend on the actual relation that we are interested in.

Now that we know how we will get test cases from a specification, we can make the connection between the test execution in the informal world and the reasoning about conformance in the formal world. We do that by making the connection between **conforms-to** and **passes**.

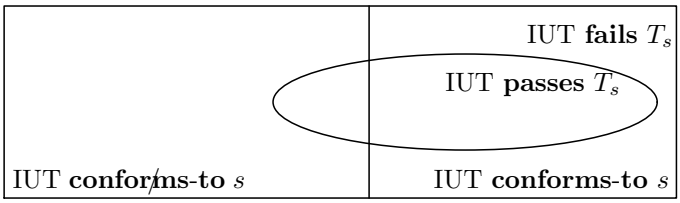


Figure 2.3: Correct and incorrect implementations pass and fail tests

Given a specification s, an implementation IUT and a test suite T_s we have that either IUT **conforms-to** s or IUT **conforms-to** s and either IUT **passes** T_s or IUT **fails** T_s. We illustrate this in Figure 2.3. In this figure the box represents all possible implementations. Those implementations that **conforms-to** s are at the right hand side of the vertical line that divides the box. Those that **conforms-to** s are at the left. Those implementations that **pass** the test suite are inside the ellipse, those outside if **fail** it. We now have two "problematic" areas in the box: we have implementations that **fail** even though they conform, and we have implementations that **pass** even though they do not conform. We call a test suite *sound* if the only implementations that **fail** are those that do not conform (however, some non-conforming implementations may **pass**). This is a very important property because when it holds each test execution failure is a symptom of a genuine error, i.e. there are no "*false negatives*". Formally:

$$\text{IUT } \textbf{conforms-to } s \Rightarrow \text{IUT } \textbf{passes } T_s \tag{2.6}$$

We call a test suite *exhaustive* if all implementations that do not conform do not **pass**, i.e. if an implementation does **pass** we are guaranteed that it conforms (however, also implementations that do conform may **fail**). Formally:

$$\text{IUT } \textbf{conforms-to } s \Leftarrow \text{IUT } \textbf{passes } T_s \tag{2.7}$$

We call a test suite *complete* if it is both sound and exhaustive, i.e. the test suite perfectly distinguishes between implementations that conform and those that do not conform. Formally:

$$\text{IUT } \textbf{conforms-to } s \Leftrightarrow \text{IUT } \textbf{passes } T_s \tag{2.8}$$

Completeness is a very strong requirement. We may need a very large (possibly infinite) test suite to realise this; executing an infinite test suite is not a very practical thing to do. In the figure, the test suite is complete when the ellipse coincides with the right part of the box.

To do real testing we have to instantiate the model with a "real" relation **imp**, for which we need a test derivation algorithm (and corresponding verdict functions). When we want to show that the test suites that we derive for a particular relation **imp** are sound or exhaustive we can do the necessary mathematical reasoning in the formal world, reasoning about models of implementations. For soundness resp. exhaustiveness we have to prove, on the level of models, the left-to-right resp. the right-to-left implication of:

$$\forall i \in MODS : i \ \mathbf{imp} \ s \Leftrightarrow \forall t \in T : v_t(obs(t, i)) = \mathbf{pass} \tag{2.9}$$

If we have proven completeness on the level of models, and we have reason to believe that the test hypothesis holds, then we obtain as consequence that we can decide on conformance of an implementation with respect to a specification by means of a testing procedure.

To conclude our discussion of the framework we give in Table 2.1 an overview of the elements that we have seen.

physical ingredients:	
black-box implementation	IUT belongs to $IMPS$
execution of a test case	$\text{EXEC}(t, \text{IUT})$
formal ingredients:	
specification	$s \in SPECS$
model of implementation	$i_{\text{IUT}} \in MODS$
implementation relation	**imp**
test case	$t \in TESTS$
observations	OBS
model of execution of a test case	$obs : TESTS \times MODS \to \mathcal{P}(OBS)$
verdict functions	$v_t : \mathcal{P}(OBS) \to \{\mathbf{pass}, \mathbf{fail}\}$
test derivation algorithm	$gen_{\mathbf{imp}} : SPECS \to \mathcal{P}(TESTS)$
assumptions:	
test hypothesis	for all IUT some i_{IUT} models IUT $obs(t, i_{\text{IUT}})$ models $\text{EXEC}(t, \text{IUT})$
prove obligations:	
soundness	$gen_{\mathbf{imp}}$ is sound for any $s \in SPECS$
exhaustiveness	$gen_{\mathbf{imp}}$ is exhaustive for any $s \in SPECS$

Table 2.1: The ingredients of the formal testing framework

Extensions Several extensions have been made to this framework. Two of them are important to us. The first enhances the framework to formally deal

with the test context (as discussed in the previous chapter on page 8). We will come back to this in Chapter 5. The other enhances the framework to formally deal with test purposes to guide the test derivation such that the resulting tests trigger the implementation to exhibit a particular behaviour of interest. We will come back to this in Chapter 4.

2.2 Instantiating the Formal Framework

We will instantiate the framework with the implementation relation that interests us: **ioco**. Before we discuss the **ioco** relation itself we present the concepts needed for its definition. In particular, we will first choose the formalisms that we will use to represent specifications, models of implementations, and tests, and we will discuss what we mean by test execution. The choices that we make here are such that they nicely provide what is needed to define **ioco**. Only then we will discuss the **ioco** implementation relation, after which we will show a test derivation algorithm for **ioco**.

2.2.1 Specifications

Labelled Transition Systems

We will use labelled transition systems as formal representation of specifications. Labelled transition systems are commonly used as a semantic model of formal specification languages, and there exists much theory on labelled transition systems.

Definition 2.2.1
A *labelled transition system* (LTS) consists of a 4-tuple $\langle S, L, T, s_0 \rangle$ with S a countable, non-empty set of states, L a countable set of labels, $T \subseteq S \times (L \cup \{\tau\}) \times S$ the transition relation, and $s_0 \in S$ the initial state. □

We use the labels in L to represent observable actions of the system and τ to represent internal actions that cannot be observed from outside the system. We use L_τ to denote $L \cup \{\tau\}$. When a state cannot do an internal (τ) action we call it *stable*. We often write $s \xrightarrow{\mu} s'$ for a transition μ from state s to state s' (this corresponds with $\langle s, \mu, s' \rangle$ of the transition relation). We interpret this as: when the system is in state s it may perform action μ and go to state s'. We represent a labelled transition system by a directed, edge-labelled graph where nodes represent states and edges represent transitions.

We call the composition of a (finite) number of transitions in a row as in $s_1 \xrightarrow{\mu_1} s_2 \xrightarrow{\cdots} s_{n-1} \xrightarrow{\mu_n} s_n$ a *computation*. It reflects both observable behaviour of the system (the sequence of actions, i.e. the transition labels) and "internal" information about system that usually can not be observed from the outside (the state in which the system is). If we take a computation and "abstract away" the information about the states we get a *trace* of the computation: a finite sequence of actions. The set of all finite sequences of actions over L is denoted by L^*, and we use ϵ to denote the empty sequence. We denote concatenation of

sequences $\sigma_1, \sigma_2 \in L^*$ as $\sigma_1 \cdot \sigma_2$. The computation above has the following trace: $\mu_1 \cdot \ldots \cdot \mu_n$. Of course, many different computations may lead to the same trace.

Definition 2.2.2
Consider labelled transition system $\langle S, L, T, s_0 \rangle$, let $s, s' \in S$, let $\mu_{(i)} \in L_\tau$, let $a_{(i)} \in L$, and $\sigma \in L^*$:

1. $s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' =_{\text{def}} \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s'$
2. $s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} =_{\text{def}} \exists s' : s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s'$
3. $s \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} =_{\text{def}}$ not $\exists s' : s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s'$
4. $s \stackrel{\epsilon}{\Longrightarrow} s' =_{\text{def}} s = s'$ or $s \xrightarrow{\tau \cdot \ldots \cdot \tau} s'$
5. $s \stackrel{a}{\Longrightarrow} s' =_{\text{def}} \exists s_1, s_2 : s \stackrel{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\epsilon}{\Longrightarrow} s'$
6. $s \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} s' =_{\text{def}} \exists s_0, \ldots, s_n : s = s_0 \stackrel{a_1}{\Longrightarrow} s_1 \stackrel{a_2}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} s_n = s'$
7. $s \stackrel{\sigma}{\Longrightarrow} =_{\text{def}} \exists s' : s \stackrel{\sigma}{\Longrightarrow} s'$
8. $s \stackrel{\sigma}{\not\Longrightarrow} =_{\text{def}}$ not $\exists s' : s \stackrel{\sigma}{\Longrightarrow} s'$

□

We will not always distinguish between a labelled transition system and its initial state: we may identify a process $p = \langle S, L, T, s_0 \rangle$ with its initial state s_0 and write, for example, $p \stackrel{\sigma}{\Longrightarrow} s_1$ instead of $s_0 \stackrel{\sigma}{\Longrightarrow} s_1$.

We now define a number of functions that we use on labelled transition systems. We use $init(p)$ to denote the outgoing transitions of a state p in a labelled transition system. We use $traces(p)$ to denote the set of traces from a state p of a labelled transition system. We use $p \,\mathbf{after}\, \sigma$ to denote the set of states that we can reach ("end up in") starting from state p by doing the transitions in σ, and we use $P \,\mathbf{after}\, \sigma$ to denote the set of states that we can reach by doing the transitions in σ when starting from a set of states P. The expression $P \,\mathbf{refuses}\, A$ is true if at least one of the states in P can not "do" (has no outgoing transition for) any of the labels in A, and also has no outgoing transition labelled τ. We use $der(p)$ to denote the set of states that we can reach from state p.

Definition 2.2.3
Consider labelled transition system $\langle S, L, T, s_0 \rangle$, let $P \subseteq S$, let $p, p' \in S$, let $A \subseteq L$, and let $\sigma \in L^*$.

1. $init(p) =_{\text{def}} \{\mu \in L_\tau \mid p \xrightarrow{\mu}\}$
2. $traces(p) =_{\text{def}} \{\sigma \in L^* \mid p \stackrel{\sigma}{\Longrightarrow}\}$
3. $p \,\mathbf{after}\, \sigma =_{\text{def}} \{p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\}$
4. $P \,\mathbf{after}\, \sigma =_{\text{def}} \bigcup \{ p \,\mathbf{after}\, \sigma \mid p \in P\}$
5. $P \,\mathbf{refuses}\, A =_{\text{def}} \exists p \in P, \forall \mu \in A \cup \{\tau\} : p \not\xrightarrow{\mu}$
6. $der(p) =_{\text{def}} \{p' \mid \exists \sigma \in L^* : p \stackrel{\sigma}{\Longrightarrow} p'\}$

□

A *failure trace* is a trace consisting of actions and *refusals*. Each action reflects a behaviour that a system is able to do, just like in an ordinary trace. Each refusal reflects behaviour that a system is not able to do. A refusal is represented by the set of actions $A \subseteq L$ that the system is not able to do (refuses to do) in a particular state. We only allow refusals in stable states

(states without outgoing τ transition). We can extend a labelled transition system with refusal transitions to make the refusals explicit. We do this by adding a self loop transition to each stable state. The self loop transition is labelled with the set of actions refused in that stable state. Formally, the refusal transition $p \xrightarrow{A} p$ for a state p in a labelled transition system means that $p \not\xrightarrow{\mu}$ for any $\mu \in (A \cup \{\tau\})$. We use $Ftraces(p)$ to denote the set of failure traces from a state p of a labelled transition system. In its definition we use $\mathcal{P}(L)$ to denote the powerset of L: the set of all subsets of L: $\mathcal{P}(L) =_{\text{def}} \{L' \mid L' \subseteq L\}$.

Definition 2.2.4
Consider labelled transition system $\langle S, L, T, s_0 \rangle$, let $P \subseteq S$, let $p, p' \in S$, let $\sigma \in L^*$, and let $A \subseteq L$.

1. $p \xrightarrow{A} p' =_{\text{def}} p = p'$ and $\forall \mu \in (A \cup \{\tau\}) : p \not\xrightarrow{\mu}$
2. $Ftraces(p) =_{\text{def}} \{\phi \in (L \cup \mathcal{P}(L))^* \mid p \stackrel{\phi}{\Longrightarrow}\}$

□

We now define the class of transition systems that we will use as the basic class for our models. We will denote it as $\mathcal{LTS}(L)$.

Definition 2.2.5
Consider labelled transition system $\langle S, L, T, s_0 \rangle$, and let $p \in S$.

1. p has *finite behaviour* if there is a natural number n such that all traces in $traces(p)$ have length smaller than n
2. p is *finite state* if the number of reachable states $der(p)$ is finite.
3. p is *deterministic* if, for all $\sigma \in L^*$, p **after** σ has at most one element. If $\sigma \in traces(p)$, then p **after** σ may be overloaded to denote this element.
4. p is *image finite* if, for all $\sigma \in L^*$, p **after** σ is finite.
5. p is *strongly converging* if there is no state of p that can perform an infinite sequence of internal transitions.
6. $\mathcal{LTS}(L)$ is the class of all image finite and strongly converging labelled transition systems with labels in L.

□

The restriction of $\mathcal{LTS}(L)$ to *image finite* and *strongly converging* transition systems makes it possible to algorithmically compute an **after** -set. As we will see in Algorithm 2.2.18 on page 39, this computation is a crucial element of our test derivation algorithm.

We let two labelled transition systems p and q interact by composing them in parallel, denoted by $p \| q$. The systems that are composed in parallel must *synchronise* on identical labels. This means that for an observable action $a \in L$ to "happen" (i.e. $p \| q \xrightarrow{a}$) both labelled transition systems must be willing to "do" action a, i.e. in both systems action a must be enabled (thus: $p \xrightarrow{a}$ and $q \xrightarrow{a}$). If that is the case, both systems do the corresponding transition at the same moment. When both systems offer more than one interaction then it is assumed that by some mysterious negotiation mechanism they will agree on a common interaction. A system can autonomously decide to "do" an internal action (a transition labelled with τ). As a consequence, a system can only make progress on observable actions in cooperation with its environment (and

autonomously, by τ transitions). If an observable action is enabled in one system but not in the other (not even after performing τ transitions), the action cannot "happen" and the latter system is said to *block* the action.

Definition 2.2.6
We define operator $\| : \mathcal{LTS}(L) \times \mathcal{LTS}(L) \to \mathcal{LTS}(L)$ by the following inference rules:

1. $p \xrightarrow{\tau} p' \ \vdash \ p \| q \xrightarrow{\tau} p' \| q$
2. $q \xrightarrow{\tau} q' \ \vdash \ p \| q \xrightarrow{\tau} p \| q'$
3. $p \xrightarrow{a} p', q \xrightarrow{a} q', a \in L \ \vdash \ p \| q \xrightarrow{a} p' \| q'$

□

This way of composing systems gives us synchronous *symmetric* communication between a system and its environment. We call it synchronous because when an interaction occurs, it occurs at exactly the same time in both composed systems. We call it symmetric because all actions are treated in the same way for both communicating partners – there is no notion of input or output, initiative or direction. We depict this in Figure 2.4.

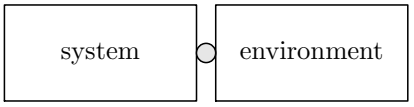


Figure 2.4: Symmetric communication between system and environment

Inputs and Outputs We have seen that with labelled transition systems we have synchronous symmetric communication between a system and its environment. There is no notion of initiative or direction, nor of input or output associated with this kind of communication.

Many real systems do not abstract from initiative and direction: they make a distinction between actions initiated by their environment, i.e. *inputs*, and actions initiated by themselves, i.e. *outputs*. To model such systems, our formalism needs to be able to express *asymmetric* communication between a system and its environment, as depicted in Figure 2.5.

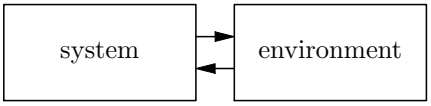


Figure 2.5: Asymmetric communication between system and environment

We define *labelled transition systems with inputs and outputs* to model systems for which the set of actions is partitioned into input actions contained in an input label set L_I and output actions contained in an output label set L_U.

Definition 2.2.7
A *labelled transition system with inputs and outputs* is a 5-tuple $\langle S, L_I, L_U, T, s_0 \rangle$ where

- $\langle S, L_I \cup L_U, T, s_0 \rangle$ is a labelled transition system in $\mathcal{LTS}(L_I \cup L_U)$;
- L_I and L_U are countable sets of input labels and output labels, respectively, which are disjoint: $L_I \cap L_U = \emptyset$.

The class of labelled transition system with inputs in L_I and outputs in L_U is denoted as $\mathcal{LTS}(L_I, L_U)$.

□

Instantiation of the formal framework We now instantiate *SPECS* in the formal framework.

For specifications we allow the use of labelled transition systems, or any formal language with a labelled transition system semantics. The actions of the transition system must be known, and, it must be possible to partition the actions into inputs and outputs, denoted by L_I and L_U respectively. No restrictions are imposed on inputs or outputs. So, we instantiate *SPECS* with $\mathcal{LTS}(L_I, L_U)$.

Notation for Labelled Transition Systems

To denote a labelled transition system we use at some places in this thesis a notation that is based on notation that is used in process-algebra [BPS01, Fok00], for example in definition 2.2.13 on page 34 and Algorithm 2.2.18 on page 39.

This notation denotes the *behaviour* of so-called *processes*, in terms of the temporal ordering of *actions*. This ordering is expressed using *behaviour-expressions*. We construct these behaviour expressions using three operators: ";", "+" and "$\sum$". Each behaviour expression that we construct with these operators corresponds to a labelled transition system, in which the actions correspond to transitions labelled with the action names.

We use the *action-prefix* operator ";" to denote that a given action is followed by subsequent behaviour: $B ::= a; B'$. For example, $p := a; p'$ denotes a transition system with state set $\{p, p'\}$, label set $\{a\}$, transition $p \xrightarrow{a} p'$, and initial state p.

The *summation* operator "+" denotes a choice between behaviours: $B ::= B_1 + B_2$. In the corresponding labelled transition system we have transitions for $B ::= B_1$ and for $B := B_2$. For example, $p_0 := a; p_a + b; p_b$ denotes a transition system with state set $\{p_0, p_a, p_b\}$, label set $\{a, b\}$, transitions $\{p_0 \xrightarrow{a} p_a, p_0 \xrightarrow{b} p_b\}$ and initial state p_0.

The operator $\sum$ generalises the *summation* operator to operate on a set of behaviours: $B := \sum B_i$. In the corresponding labelled transition system we have transitions for each $B ::= B_i$. For example, $p_0 := \sum\{x_i; p_{x_i} \mid x_i \in \{a, b\}\}$ denotes a transition system with state set $\{p_0, p_a, p_b\}$, label set $\{a, b\}$, transitions $\{p_0 \xrightarrow{a} p_a, p_0 \xrightarrow{b} p_b\}$ and initial state p_0.

We denote cycles by writing a recursive behaviour expression: $p := a; p$ denotes a state p with a self-loop $p \xrightarrow{a} p$.

2.2.2 Implementations and Implementation Models

Input-output Transition Systems

The formalism labelled transition systems with inputs and outputs allows us to distinguish between inputs and outputs. However, to allow a system to make progress autonomously on the actions that it initiates, we need something more.

The formalism of *input-output transition systems* allows us to model systems with inputs and outputs, in which outputs are initiated by the system and never refused by the environment, and inputs are initiated by the system's environment and never refused by the system. This means that the system is always prepared to perform any input action, i.e. all inputs are always enabled in all states. We say that the system is *input-enabled*.

Definition 2.2.8
An *input-output transition system* is a labelled transition system with inputs and outputs $\langle S, L_I, L_U, T, s_0 \rangle$ where all input actions are enabled in any reachable state:

$$\forall s \in der(s_0), \forall a \in L_I : s \stackrel{a}{\Longrightarrow}$$

We use $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$ to denote the class of input-output transition systems with input actions in L_I and output actions in L_U. □

Quiescence

When we let two input-output transition systems interact and make sure that the inputs of the one are the outputs of the other and vice versa, an input-output transition system can autonomously decide to perform output actions because its environment will always accept them. In states where output actions are enabled it can autonomously decide whether to continue or not. However, in (stable) states where no output actions are enabled it has to wait for the environment to provide the next input action. If that is the case, we say that the system is *quiescent* or *suspended*. A state in which no output actions are enabled is called a *quiescent state*.

We use $\delta(p)$ to denote that state p is quiescent. A state p is quiescent if $\forall \mu \in (L_U \cup \{\tau\}) : p \stackrel{\mu}{\not\longrightarrow}$. We use the special action $\delta \notin (L \cup \{\tau\})$ to represent quiescence. We can extend a labelled transition system to make the quiescence explicit by adding self loops with label δ, i.e. $p \stackrel{\delta}{\longrightarrow} p$. *Suspension traces* are traces of a system in which action δ may occur. We use $Straces(p)$ to denote the set of suspensions traces from a state p of a labelled transition system.

Definition 2.2.9
Let $p \in \mathcal{LTS}(L)$.

1. $Straces(p) \ =_{\text{def}} \ \{\phi \in (L \cup \{\delta\})^* \mid p \stackrel{\phi}{\Longrightarrow}\}$

□

In a way, quiescence can be seen as refusal of all output actions. So, we can also see a suspension trace as a failure trace where all refusals are $(L_U \cup \{\tau\})$, i.e. δ.

For input-output transition systems we use $out(p)$ to denote the set containing all output actions that are enabled in state p. If p is quiescent we add δ to $out(p)$. We use L_U^δ to denote $L_U \cup \{\delta\}$.

Definition 2.2.10
Let p be a state in a transition system, and P be a set of states, then

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

□

We use $in(p)$ to denote the set containing all input actions that are enabled in state p.

Definition 2.2.11
Let p be a state in a transition system, and P be a set of states, then

1. $in(p) =_{\text{def}} \{a \in L_I \mid p \xrightarrow{a}\}$
2. $in(P) =_{\text{def}} \bigcup\{in(p) \mid p \in P\}$

□

Suspension Automaton

From a labelled transition system with inputs, outputs and quiescence we can create a deterministic transition system of which the traces are the suspension traces of the original system, as follows. We first make quiescence explicit in the system by adding self loops with label δ, i.e. $p \xrightarrow{\delta} p$ for all quiescent states, after which we determinize the resulting automaton. The resulting deterministic transition system is referred to as a *suspension automaton.*

Definition 2.2.12

Let $p = \langle S, L_I, L_U, T, s_0\rangle \in \mathcal{LTS}(L_I, L_U)$ be a labelled transition system, with inputs and outputs, then the *suspension automaton* of p, Γ_p, is the labelled transition system $\langle S_\Gamma, L_I, L_U^\delta, T_\Gamma, q_0\rangle$, where

$$
\begin{array}{lll}
S_\Gamma & =_{\text{def}} & \mathcal{P}(S) \setminus \{\emptyset\} \\
L_U^\delta & =_{\text{def}} & L_U \cup \{\delta\} \\
T_\Gamma & =_{\text{def}} & \{q \xrightarrow{a} q' \mid a \in L_I \cup L_U,\ q, q' \in S_\Gamma,\ q' = \{s' \in S \mid \exists s \in q : s \overset{a}{\Longrightarrow} s'\}\} \\
 & & \cup\{q \xrightarrow{\delta} q' \mid q, q' \in S_\Gamma,\ q' = \{s \in q \mid \delta(s)\}\} \\
q_0 & =_{\text{def}} & \{s' \in S \mid s_0 \overset{\epsilon}{\Longrightarrow} s'\}
\end{array}
$$

□

We consider δ to be an output action of a suspension automaton, i.e. it has L_I as inputs and $L_U \cup \{\delta\}$ as outputs. A suspension automaton is deterministic. In the suspension automaton $\xrightarrow{\sigma}$ and $\overset{\sigma}{\Longrightarrow}$ coincide. The traces of a suspension automaton Γ_p are identical to the suspension traces of the original system p. Each state $q \in S_\Gamma$ of suspension automaton Γ_p coincides with (is) a set of states $Q \subseteq S$ of the original system p: the set of states p **after** σ that is reached by σ, a suspension trace of p. Such a state has (outgoing) transitions $q \xrightarrow{a} q'$ for each input label $a \in in(q)$ and each output (or quiescence) label $a \in out(q)$. Furthermore σ is a trace of Γ_p iff $out(\,\Gamma_p$ **after** $\sigma\,) \neq \emptyset$, because the set of states P' that we reach by a valid suspension trace from p either has at least one state with an outgoing output action, or contains at least one quiescent state. In either case $out(P') \neq \emptyset$.

Instantiation of the formal framework We now instantiate *MODS* and *IMPS* in the formal framework.

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, we instantiate *MODS* with $\mathcal{IOTS}(L_I, L_U)$.

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

2.2.3 Tests

We now define the formalisation of test cases. A test case specifies the experiment that we want to conduct on an implementation. We use a special kind of input-output transition system for test cases. It will synchronise with the implementation, and it will use δ to represent the observation of quiescence. We want tests to be finite (we will never be able to run an infinite test in finite time). We want to have maximal control over the testing. Therefore, we want tests to be deterministic. A test should not allow the choice between multiple input actions. However, the test should be input-enabled for all outputs of the implementation (to avoid blocking the implementation when it wants to provide output). Note that we do not regard quiescence to be an output of the implementation – it is something that we observe during test execution, typically when a timer expires. As a consequence each state of a test case is either a sink state (accepting all outputs of the implementation) or a state that offers exactly one input to the implementation (and accepts all outputs of the implementation), or a state that accepts all outputs of the implementation and quiescence. Each test execution should result in a verdict, **pass** or **fail**, which we achieve by labelling the sink states with **pass** and **fail**.

Definition 2.2.13
A *test case* t for an implementation with inputs in L_I and outputs in L_U is an input-output transition system $\langle S, L_U, L_I \cup \{\delta\}, T, s_0 \rangle \in \mathcal{IOTS}(L_U, L_I \cup \{\delta\})$ such that:

1. t is deterministic and is finite state
2. S contains special states **pass** and **fail**, **pass** $\neq$ **fail**, with
   $\textbf{pass} := \sum\{x; \textbf{pass} \mid x \in L_U \cup \{\delta\}\}$
   $\textbf{fail} := \sum\{x; \textbf{fail} \mid x \in L_U \cup \{\delta\}\}$
3. t has no cycles except those in states **pass** and **fail**
   (formally: for $\sigma \in (L \cup \{\delta\})^* \setminus \{\epsilon\} : q \stackrel{\sigma}{\Longrightarrow} q$ implies $q =$ **pass** or $q =$ **fail**)
4. for any state $p \in S$ of the test case
   either $init(p) = \{a\} \cup L_U$ for some $a \in L_I$,
   or $init(p) = L_U \cup \{\delta\}$

The class of test cases for implementations with inputs L_I and outputs L_U is denoted as $\mathcal{TTS}(L_U, L_I)$.

A test suite T is a set of test cases: $T \subseteq \mathcal{TTS}(L_U, L_I)$.

□

Instantiation of the formal framework We instantiate *TESTS* of the formal framework with $\mathcal{TTS}(L_U, L_I)$.

Bibliographic note Initial publications of the theory that we summarise in this chapter defined test cases that were not input enabled [Tre96]. Only relatively recently that initial theory has been enhanced with the definition of input-enabled test cases, and with corresponding changes to the test derivation algorithm [Tre08].

2.2.4 Test Execution, Observations and Verdicts

When a test is executed outputs of the specification (i.e. outputs of the test case) are inputs for the implementation and vice versa. To reduce possible confusion we will use the term *stimulus* for an output of a test case (i.e. an input to the implementation) and *response* for an output of the implementation (i.e. an input to the test case).

Execution of a test case on an implementation is modelled by parallel composition of the test case and the implementation, where inputs of the test case synchronise with outputs of the implementation and vice versa. *Note that in case of quiescence the test case and the implementation synchronise on δ.* Test execution takes place until the test case reaches one of its sink states. Reaching a sink state is guaranteed by the special structure of a test case. The verdict of a test run is given by the label of the sink state (**pass** or **fail**).

Definition 2.2.14

1. A *test run* of test case $t \in \mathcal{TTS}(L_U, L_I)$ and implementation $i \in \mathcal{IOTS}(L_I, L_U)$ is a trace of the synchronous parallel composition $t \| i$ leading to one of the sink states of t:

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

2. Implementation i *passes* test case t if all their test runs lead to a sink state of t with label **pass**:

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

3. Implementation i *passes* test suite T if it passes all test cases in T:

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

If i does not pass test suite T, it fails:

$$i \;\mathbf{fails}\; T \;=_{\text{def}}\; \exists t \in T : i \;\mathbf{pas\not{s}es}\; t$$

□

Instantiation of the formal framework We can now instantiate OBS, obs, and v_t in the formal framework

As observations we can use logs of actions, i.e. traces over $L \cup \{\delta\}$, so OBS is instantiated with $(L \cup \{\delta\})^*$.

We can define the observation function using the test runs we have defined. It restricts OBS to those observations that can be obtained from test runs for given t and i:

$$obs(t, i) \ =_{\text{def}} \ \{\sigma \in (L \cup \{\delta\})^* | \ \sigma \text{ is a test run of } t \text{ and } i\} \tag{2.10}$$

An implementation i passes a test case t if all their test runs lead to a state labelled **pass** of t. We rephrase this in terms of the testing framework by defining the verdict functions that assign a verdict to set of observations $O \subseteq OBS$ as:

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

2.2.5 Implementation Relation

Two elements of the formal framework now remain to be instantiated: the relation **imp** and the corresponding test derivation algorithm $gen_{\textbf{imp}}$. We have already defined $TESTS$, which imposes constraints on $gen_{\textbf{imp}}$.

As said in the introduction to this chapter, we will use implementation relation **ioco**. Informally, **ioco** requires that all outputs produced by the implementation during test runs can be predicted by the specification. Also all absence of output (quiescence) of the implementation must be predicted by the specification.

Definition 2.2.15
i **ioco** $s \ =_{\text{def}} \ \forall \sigma \in Straces(s) : out(\, i \textbf{ after } \sigma \,) \subseteq out(\, s \textbf{ after } \sigma \,)$

□

Instantiation of the formal framework We instantiate **imp** of the formal framework with **ioco**.

Generalisations and Extensions

We discuss a few of the generalisations and extensions of **ioco** that have been proposed in the literature. Section 4.2 of [Tre08] contains a more elaborate overview.

ioco$_{\mathcal{F}}$ Relation **ioco** is a specific instance of a family of implementation relations **ioco**$_{\mathcal{F}}$.

Definition 2.2.16
i **ioco**$_{\mathcal{F}}$ $s \ =_{\text{def}} \ \forall \sigma \in \mathcal{F} : out(\, i \textbf{ after } \sigma \,) \subseteq out(\, s \textbf{ after } \sigma \,)$

□

The relations in $\mathbf{ioco}_{\mathcal{F}}$ differ in $\mathcal{F}$: the traces that they consider for the test runs.

One such difference is whether or not the traces can contain (multiple occurrences of) quiescence. Another difference is whether all possible traces are considered, or only those that are present in the specification.

If only those traces are considered that are present in the specification, then the implementation is free to implement additional functionality as long as it is triggered by an input that is not enabled at that point in the specification. During testing this additional functionality will not be reached because the tester will never apply the stimulus that triggers it.

If all possible traces are considered the relation is more strict. In that case all behaviour of the implementation after an input that was not enabled at that point in the specification will result in failures, due to the definition of *out*: if $\sigma \in \mathit{Straces}(s)$ and $a \in L_I$ and $\sigma \cdot a \notin \mathit{Straces}(s)$ then $s \,\mathbf{after}\, \sigma \cdot a = \emptyset$ and thus $\mathit{out}(\, s \,\mathbf{after}\, \sigma \cdot a\,) = \emptyset$, whereas, assuming $\sigma \cdot a \in \mathit{Straces}(i)$ then $P = i \,\mathbf{after}\, \sigma \cdot a \neq \emptyset$, which means that there is at least one state $p \in P$. State p will either be quiescent or non-quiescent. If state p is quiescent, $\delta \in \mathit{out}(P) \neq \emptyset$; if state p is non-quiescent, there is at least one output action $x \in L_U$ that is enabled in p, and thus $x \in \mathit{out}(P) \neq \emptyset$. In both cases we have that $\mathit{out}(\, i \,\mathbf{after}\, \sigma \cdot a\,) \neq \emptyset$ whereas $\mathit{out}(\, s \,\mathbf{after}\, \sigma \cdot a\,) = \emptyset$ and thus $\mathit{out}(\, i \,\mathbf{after}\, \sigma \cdot a\,) \not\subseteq \mathit{out}(\, s \,\mathbf{after}\, \sigma \cdot a\,)$.

In the remainder of this thesis we do not consider the general family of implementation relations $\mathbf{ioco}_{\mathcal{F}}$, but only its specific instance **ioco** for which $\mathcal{F} = \mathit{Straces}(s)$.

$\mathbf{mioco}_{\mathcal{F}}$ In [Hee98] an implementation relation is presented that generalises $\mathbf{ioco}_{\mathcal{F}}$: $\mathbf{mioco}_{\mathcal{F}}$. Relation $\mathbf{ioco}_{\mathcal{F}}$ (implicitly) deals with a single communication channel between a system and its environment (or, to be more precise, with two uni-directional channels), such that a property like quiescence is perceived as a global property of a system. In contrast, $\mathbf{mioco}_{\mathcal{F}}$ allows multiple communication channels between a system and its environment, where quiescence is a per-channel property (i.e. the implementation may be quiescent on one channel, and simultaneously producing output on another one). In addition, $\mathbf{ioco}_{\mathcal{F}}$ assumes (depends on) input-completeness of the implementation, whereas $\mathbf{mioco}_{\mathcal{F}}$ also allows *input-suspension*, i.e. allows the implementation to refuse input, again per channel.

Where for $\mathbf{ioco}_{\mathcal{F}}$ implementations are modelled as input-output transition system and the label set L is partitioned in input label set L_I and output label set L_U, for $\mathbf{mioco}_{\mathcal{F}}$ implementations are modelled as *multi input-output transition systems* in which L_I is partitioned into multiple disjoint sets of input labels L_I^j and L_U is partitioned into multiple sets of output labels L_U^k, each of which corresponds to a uni-directional input or output communication channel between the implementation and its environment. Each of the input and output label sets has its own special action δ_i^j or δ_u^k to represent input suspension resp. quiescence on the corresponding channel.

In [Hee98] it is shown that $\mathbf{ioco}_{\mathcal{F}}$ is a special case of $\mathbf{mioco}_{\mathcal{F}}$. Also a test derivation algorithm is given for $\mathbf{mioco}_{\mathcal{F}}$, of which soundness and exhaustiveness

is proved.

uioco Relation **uioco**, presented in [vdBRT04] was designed to better handle specifications that are not input-enabled and therefore *underspecified*. When a specification is not input-enabled and contains non-determinism it may occur that we get test failures that were not intended. This may happen when during test execution, at a point where the current tester S state consists of multiple specification states, we chose to apply an input a that is enabled in some, but not all states in S. For those states in S for which a is not enabled, we effectively did not specify how the implementation should behave after input a. However, when judging the behaviour of the implementation after a, **ioco** only considers the behaviour in states S **after** a as correct. This ignores the possibility that the implementation might have been in a state that corresponds with a state in S in which a was not enabled, and because of that produced output, or did not produce output, in a way that is not accounted for in the specification – leading to an unexpected observation, and a **fail** verdict.

With relation **uioco** this is resolved by not using the *Straces* of the specification for testing, but a subset of it, the *Utraces* that only contains those input actions that are not underspecified.

Definition 2.2.17
Consider input-output transition system $i \in \mathcal{IOTS}(L_i, L_U)$, and labelled transition system with inputs and outputs $s \in \mathcal{LTS}(L_I, L_U)$.

1. $Utraces(s) =_{\text{def}} \{\sigma \in Straces(s) \mid \forall \sigma_1, \sigma_2 \in L^*, a \in L_I :$
   $\sigma = \sigma_1 \cdot a \cdot \sigma_2$ implies not s **after** σ_1 **refuses** $\{a\}\}$
2. i **uioco** $s =_{\text{def}} i$ **ioco**$_{Utraces(s)}$ s

□

tioco$_M$, rtioco For testing with (real-)time, various extensions of **ioco** have been proposed, of which we mention **tioco$_M$**, presented in [BB04], and **rtioco**, presented in [LMN05]. For an overview of relations for timed testing, see [ST08].

sioco Relation **sioco**, presented in [FTW05] extends **ioco** for testing in a setting where actions are parameterised with data, and where the data is treated in a symbolic way to avoid state explosion during test derivation. It does not alter **ioco**, but only gives a presentation of the relation in case data variables and parameters are involved.

2.2.6 Test Derivation

There are several algorithms to derive tests for **ioco$_\mathcal{F}$** or **ioco** in the literature. Most of these derive the non-input enabled test cases of [Tre96]. We are aware of only one that derives input-enabled test cases.

For non-input-enabled test cases We first discuss the older algorithms that derive non-input-enabled test cases. They differ in two aspects. They differ in whether they derive tests directly from the specification, other via an

intermediate structure, and they differ in whether the set of traces that they consider (as denoted by $\mathcal{F}$) is made explicit in the algorithm.

We are aware of two algorithms to derive tests for $\mathbf{ioco}_{\mathcal{F}}$. Both these algorithms make the set of traces that are considered, denoted by $\mathcal{F}$, explicit in the algorithm. Both these algorithms do not derive tests for arbitrary $\mathcal{F}$ but only for suspension traces of the specification ($\mathcal{F} = \mathit{Straces}(s)$). Algorithm 6.2 of [Tre96] does not derive the tests directly from the (traces of the) specification but from (traces of) an intermediate structure, the *suspension automaton* as defined in definition 2.2.12. The algorithm given in Figure 7.9 of [Hee98] derives tests directly from the specification, i.e. it does not need the intermediate suspension automaton. Algorithm 5.3.2 of [Tre02] derives tests directly for **ioco**, directly from the specification. In this algorithm $\mathcal{F}$ is not explicitly present.

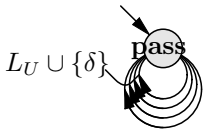
For input-enabled test cases Algorithm 1 of [Tre08] derives input-enabled tests directly for **ioco**, directly from the specification. In this algorithm $\mathcal{F}$ is not explicitly present. We paraphrase this algorithm below in algorithm 2.2.18.

Algorithm 2.2.18
Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification, and let S be a non-empty set of states with initially $S = s \,\mathbf{after}\, \epsilon$. A test case $t \in \mathcal{TTS}(L_U, L_I)$ is obtained from S by a finite number of recursive applications of one of the following three nondeterministic choices (rules). The resulting test case has a tree structure. Initially the tree consists of just the root node which acts as a placeholder to be expanded. Each rule expands a placeholder node, but may also add new placeholders node. Below we use white (non-filled) nodes to represent placeholders and dark (filled) nodes to represent placeholders that have been expanded. In the rules we use a process-algebraic notation to denote the construction of the tree that is explained where we use it.

Initially, the tree consists of just the root node which acts as a placeholder to be expanded. This corresponds to a transition system with a single state that is not **pass** or **fail**.

The rules are:

1. (* terminate the test case; stop the recursion in the algorithm *)
   $t :=$ **pass**

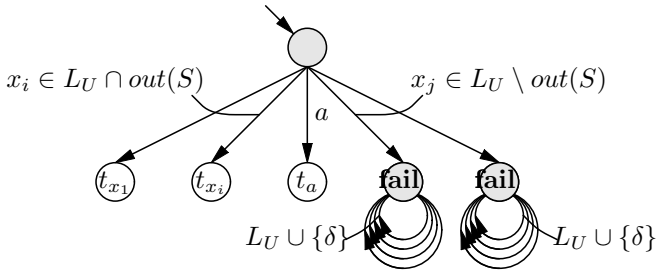


   This rule turns a placeholder t into a sink state **pass**. In this state any output from the implementation (including quiescence) is accepted.

2. (* give a next input to the implementation, but be prepared to accept output *)

$$\begin{aligned} t := \quad & a; t_a \\ + \quad & \sum\{x_i; t_{x_i} \mid x_i \in L_U \wedge x_i \in out(S)\} \\ + \quad & \sum\{x_j; \mathbf{fail} \mid x_j \in L_U \wedge x_j \notin out(S)\} \end{aligned}$$

where $a \in L_I$, S **after** $a \neq \emptyset$ (i.e. $a \in in(S)$)
and t_a is obtained by recursively applying the algorithm on S **after** a
and for each $x_i \in out(S)$, t_{x_i} is obtained by recursively applying the algorithm for the set of states S **after** x_i .

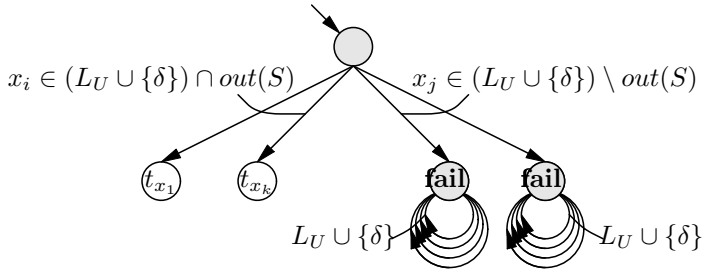


This rule extends the tree by turning a placeholder t into a non-leaf node by creating from it a whole set of edges: one for the input that we want to give, and one for each output label (but not for quiescence). The edge for the input is labelled with an input action a, arbitrarily chosen from the input actions enabled at this point. This edge points to a newly created placeholder t_a. Those edges that correspond to output actions that are not present in (expected by) the specification at that point lead to a sink state **fail**. Those edges that correspond to output actions that are present in the specification lead to newly created placeholders. Note that here we only accept immediate "pending" output actions – we do not accept quiescence, because during test execution we do not intend to wait at this point until quiescence is detected.

3. (* check the next output of the implementation *)

$$\begin{aligned} t := \quad & \sum\{x_i; t_{x_i} \mid x_i \in (L_U^\delta) \wedge x_i \in out(S)\} \\ + \quad & \sum\{x_j; \mathbf{fail} \mid x_j \in (L_U^\delta) \wedge x_j \notin out(S)\} \end{aligned}$$

where t_{x_i} is obtained by recursively applying the algorithm on S **after** x .



This rule extends the tree by turning a placeholder t into a non-leaf node by creating from it a whole set of edges, one for each output label (including quiescence). Those edges that correspond to output actions or quiescence states that are not present in (expected by) the specification at that point lead to a sink state **fail**. Those edges that correspond to output actions (or quiescent states) that are present in the specification lead to newly created placeholders.

□

Each run of algorithm 2.2.18 produces a test case. Each test case so produced is sound by construction, by the following reasoning. (Proofs of soundness and exhaustiveness for derivation of non-input-enabled test cases can be found in [Tre96]. We are not aware of publications that give these proofs for the case of input-enabled test cases.) The tests only lead to **fail** verdicts when outputs are checked. In every step where outputs are checked (say, after performing trace σ) the only outputs x that lead to **fail** states are those for which $x \notin out(\, s \textbf{ after } \sigma \,)$. The other outputs lead to **pass** or test deeper. So, the only way for an implementation to fail the test is to produce an output $x \notin out(\, s \textbf{ after } \sigma \,)$. However, in that case the implementation does not conform to **ioco** because the produced output $x \in out(\, i \textbf{ after } \sigma \,) \not\subseteq out(\, s \textbf{ after } \sigma \,)$ because $x \notin out(\, s \textbf{ after } \sigma \,)$.

Each run of the algorithm contains a number of non-deterministic choices: in each recursive step (i.e. for each placeholder) a rule has to be chosen. In addition, when applying a stimulus in rule 2, a stimulus has to be chosen from L_I. By choosing differently in different runs of the algorithm we obtain different test cases. If we run the algorithm repeatedly and choose differently in a systematic way we eventually end up with a (possible infinite) test suite containing all possible tests that can be derived from the specification.

It has been shown in [Tre96] that the resulting (possibly infinite) test suite (of non-input enabled test cases) is exhaustive.

The fact that a possibly infinite test suite is exhaustive may seem to be of little practical value. However, the fact that the test suite is exhaustive, even if only "in the limit" demonstrates that the test derivation algorithm does consider all possible test cases that "in the limit" find (detect) all possible errors. This means that there are no inherent "blind spots" in the algorithm.

Definition 2.2.19
Let $s \in \mathcal{LTS}(L_I \cup L_U)$ be a specification and T_s be the set of all test cases that can be derived from s with algorithm 2.2.18, then we define $gen_{\mathbf{ioco}} : \mathcal{LTS}(L_I, L_U) \rightarrow \mathcal{P}(\mathcal{TTS}(L_U, LI))$ as follows:
$gen_{\mathbf{ioco}} =_{\text{def}}$ a test derivation function such that $gen_{\mathbf{ioco}}(s) \subseteq T_s$

□

Instantiation of the formal framework We instantiate $gen_{\mathbf{imp}}$ of the formal framework with $gen_{\mathbf{ioco}}$.

2.2.7 Overview

To conclude our discussion of how we instantiate the framework we present in Table 2.2 on the current page an updated version of Table 2.1 on page 26, in which the instantiations that we have chosen are shown.

physical ingredients:	
black-box implementation	IUT can be any computer system or program which can can be modelled as input-output transition system, where inputs and outputs can be mapped one-to-one on L_I resp. L_U and inputs can always occur
execution of a test case	EXEC(t, IUT): physical execution of test case t on IUT; may involve multiple runs, e.g. to cater for non-determinism
formal ingredients:	
specification	$s \in \mathcal{LTS}(L_I, L_U)$
model of implementation	$i_{\text{IUT}} \in \mathcal{IOTS}(L_I, L_U)$
implementation relation	**ioco**
test case	$t \in \mathcal{TTS}(L_U, L_I)$
observations	$(L \cup \{\delta\})^*$
model of execution of a test case	$obs : \mathcal{TTS}(L_U, L_I) \times \mathcal{IOTS}(L_I, L_U) \rightarrow \mathcal{P}((L \cup \{\delta\})^*)$
verdict functions	$v_t : \mathcal{P}((L \cup \{\delta\})^*) \rightarrow \{\textbf{pass}, \textbf{fail}\}$
test derivation algorithm	$gen_{\textbf{ioco}} : \mathcal{LTS}(L_I, L_U) \rightarrow \mathcal{P}(\mathcal{TTS}(L_U, L_I))$
assumptions:	
test hypothesis	for all IUT some i_{IUT} models IUT $obs(t, i_{\text{IUT}})$ models EXEC(t, IUT)
prove obligations:	
soundness	$gen_{\textbf{ioco}}$ is sound for any $s \in \mathcal{LTS}(L_I, L_U)$
exhaustiveness	$gen_{\textbf{ioco}}$ is exhaustive for any $s \in \mathcal{LTS}(L_I, L_U)$

Table 2.2: Our instantiation of the ingredients of the formal testing framework

2.3 Formal Framework for Observation Objectives

We now extend the framework of Section 2.1 with *observation objectives*.

An observation objective describes the observations that we wish to see from the implementation during a test. Whether an implementation is able to show these observations is called *exhibition*: an implementation *exhibits* an observation objective if it has the possibility to show the observations described in the observation objective. The idea is that this notion of exhibition is orthogonal to correctness.

We assume $TOBS$ to be the set of all observation objectives, and thus observation objective $e \in TOBS$. The relation **exhibits** $\subseteq IMPS \times TOBS$ relates an observation objective with all implementations that are able to *exhibit* that observation objective. Now we would like to have a formal relation between e and IUT, but as we have seen in the framework for conformance testing, the implementation is not a formal object. Again we link the informal, experimental world and the formal world by making the assumption of the testing hypothesis, and we formally express the relation between exhibition of observation objectives by (models of) implementations as the *reveal relation* **rev** $\subseteq MODS \times TOBS$. We can now mirror the informal relation **exhibits** between $IMPS$ and $TOBS$ in the informal world by the (formal) relation **rev** between $MODS$ and $TOBS$ in the formal world, as depicted in Figure 2.6, and we say that IUT satisfies an observation objective e, i.e. IUT **exhibits** e, if and only if relation **rev** holds between i_{IUT} and e: i_{IUT} **rev** e.

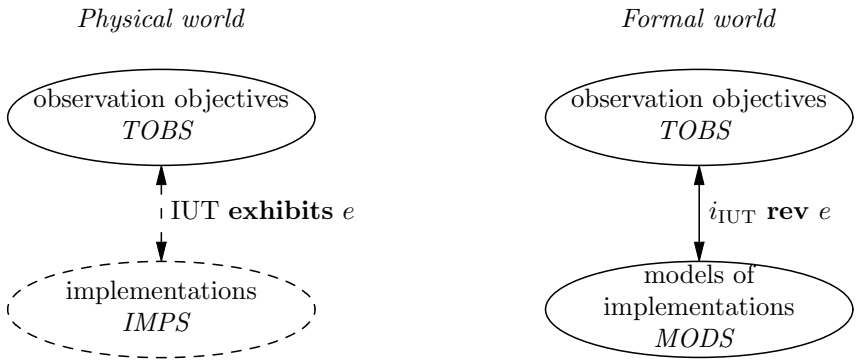


Figure 2.6: Relating implementations and implementation models with observation objectives.

To decide whether an implementation exhibits an observation objective we use testing, and interpret the observations obtained by the experiments. Like in the framework for conformance testing, we associate a verdict with the result of executing a test on an implementation. We formalise this using a family of *hit functions* $H_e : \mathcal{P}(OBS) \to \{\mathbf{hit}, \mathbf{miss}\}$. Here **hit** expresses that we have found evidence during experimenting that the implementation is able to exhibit a given

observation objective. We can now define what it means to hit an observation objective by a test, where t_e is a test case related to an observation objective e, i.e. developed based on e.

$$\text{IUT } \textbf{hits } e \textbf{ by } t_e \ =_{\text{def}} \ H_e(\text{EXEC}(t_e, \text{IUT})) = \textbf{hit} \tag{2.11}$$

This is extended to hitting an observation objective by a test suite T_e (note how this differs from the extension for passing a test suite, cnf. Equation 2.4)

$$\text{IUT } \textbf{hits } e \textbf{ by } T_e \ =_{\text{def}} \ H_e(\cup\{\text{EXEC}(t, \text{IUT}) \mid t \in T_e\}) = \textbf{hit} \tag{2.12}$$

An implementation misses a test suite T_e if it does not hit:

$$\text{IUT } \textbf{misses } e \textbf{ by } T_e \ =_{\text{def}} \ \neg(\text{IUT } \textbf{hits } e \textbf{ by } T_e) \tag{2.13}$$

We now make a connection between **exhibits** and **hits**. Analogous to conformance testing, we can introduce notions of completeness, exhaustiveness and soundness. We call a test suite *e-exhaustive* when it only can detect non-exhibiting implementations. Formally:

$$\text{IUT } \textbf{exhibits } e \textbf{ by } T_e \Rightarrow \text{IUT } \textbf{hits } e \textbf{ by } T_e \tag{2.14}$$

We call a test suite *e-sound* when it only can detect exhibiting implementations. Formally:

$$\text{IUT } \textbf{exhibits } e \textbf{ by } T_e \Leftarrow \text{IUT } \textbf{hits } e \textbf{ by } T_e \tag{2.15}$$

We call a test suite *e-complete* when it can distinguish among all exhibiting and non-exhibiting implementations. Formally:

$$\text{IUT } \textbf{exhibits } e \textbf{ by } T_e \Leftrightarrow \text{IUT } \textbf{hits } e \textbf{ by } T_e \tag{2.16}$$

To reason whether a test suite is able to challenge an implementation to exhibit an observation objective and is able to detect all exhibiting implementations, we have to show e-soundness and e-completeness of such a test suite, i.e. we have to prove, on the level of models, the left-to-right resp. right-to-left implication of:

$$\forall i \in \mathit{MODS} : i \textbf{ rev } e \Leftrightarrow H_e(\cup\{\mathit{obs}(t_e, i) \mid t_e \in T_e\}) = \textbf{hit} \tag{2.17}$$

Finally, we have to decide which relation we want between correctness and exhibition.

For a practical approach to conformance testing using exhibition we choose our observation objectives e such that

$$\{i \mid i \textbf{ rev } e\} \cap \{i \mid i \textbf{ imp } s\} \neq \emptyset \tag{2.18}$$

This is visualised in Figure 2.7. (This scenario for the intersection of i **rev** e and i **passes** T is, together with three other possible scenarios, discussed in greater detail in [dV01].) Here i **imp** s is the set of all conforming models of the implementation, i **passes** T is the set of all models that pass a sound test suite of T, and i **rev** e is the set of models that exhibit observation objective

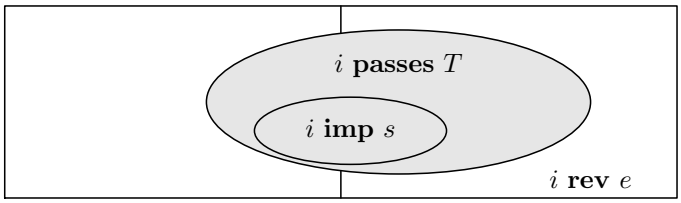


Figure 2.7: Practical approach towards combining exhibition and conformance

e (this latter set corresponds to the set of IUT **hits** e **by** T_e with T_e being an e-complete test suite).

We have most confidence in the correctness of an implementation that conforms and exhibits (verdict $\langle \mathbf{pass}, \mathbf{hit} \rangle$). However, we might still find an implementation that is correct, but does not exhibit (which corresponds to the notion of the **inconclusive** verdict [ISO91]), or an implementation that is incorrect and exhibits.

From such an observation objective e and formal specification s we derive a test suite $T_{s,e}$ that is e-complete and sound. After execution of $T_{s,e}$ we reject an implementation that gives a **fail** verdict. We have an increased confidence in the correctness when the test result evaluates to a $\langle \mathbf{pass}, \mathbf{hit} \rangle$ verdict. An experiment that evaluates to a $\langle \mathbf{pass}, \mathbf{miss} \rangle$ verdict has not found any evidence of non-conformance of the implementation, but has also not detected any behaviour that supports our confidence in the correctness of the implementation.

To conclude our discussion of the extension of the framework for conformance testing with we give an overview of the elements that we have seen in Table 2.1.

2.4 Instantiating the Formal Framework for Observation Objectives

We now instantiate the framework for exhibition presented in Section 2.3 with the **ioco** theory of Section 2.2.

2.4.1 Reveal Relations

We instantiate the framework of Section 2.3 with the **ioco** theory. We inherit $TESTS = \mathcal{TTS}(L_U, L_I)$, $OBSi = (L \cup \{\delta\})^*$, and $MODS = \mathcal{IOTS}(L_I, L_U)$.

Singular observation objectives We first consider *singular observation objectives*. A singular observation objective can be exhibited by one observation of a test case execution.

We take as specification of the observation objective a set of traces from L_δ^*. So $TOBS = \mathcal{P}(L_\delta^*)$ and we instantiate $\mathbf{rev} \subseteq \mathcal{IOTS}(L_I, L_U) \times ((L_I \cup L_U \cup \{\delta\})^*)$.

We define the *reveal input output singular* relation **rios** for the singular case. This relation relates formally all models of implementations that are potentially

physical ingredients:	
black-box implementation	IUT belongs to *IMPS*
execution of a test case	$\mathrm{EXEC}(t, \mathrm{IUT})$
formal ingredients:	
specification	$s \in SPECS$
model of implementation	$i_{\mathrm{IUT}} \in MODS$
implementation relation	**imp**
observation objective	$e \in TOBS$
reveal relation	**rev**
test case	$t \in TESTS$
observations	OBS
model of execution of a test case	$obs : TESTS \times MODS \rightarrow \mathcal{P}(OBS)$
verdict functions	$v_t : \mathcal{P}(OBS) \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$
hit functions	$H_e : \mathcal{P}(OBS) \rightarrow \{\mathbf{hit}, \mathbf{miss}\}$
test derivation algorithm	$gen_{\mathbf{imp}} : SPECS \rightarrow \mathcal{P}(TESTS)$
assumptions:	
test hypothesis	for all IUT some i_{IUT} models IUT $obs(t, i_{\mathrm{IUT}})$ models $\mathrm{EXEC}(t, \mathrm{IUT})$
prove obligations:	
soundness	$gen_{\mathbf{imp}}$ is sound for any $s \in SPECS$
exhaustiveness	$gen_{\mathbf{imp}}$ is exhaustive for any $s \in SPECS$
e-soundness	$gen_{\mathbf{imp}}$ is e-sound for any $e \in TOBS$ for which $\{i \mid i \ \mathbf{rev}\ e\} \cap \{i \mid i \ \mathbf{imp}\ s\} \neq \emptyset$
e-exhaustiveness	$gen_{\mathbf{imp}}$ is e-exhaustive for any $e \in TOBS$ for which $\{i \mid i \ \mathbf{rev}\ e\} \cap \{i \mid i \ \mathbf{imp}\ s\} \neq \emptyset$

Table 2.3: The ingredients of the formal testing framework extended with exhibition testing

able to exhibit a singular test objective to that observation objective. A model of an implementation exhibits the singular test objective if one of its suspension traces is an element of the observation objective. So a **hit**-function H_e for a singular observation objective evaluates to **hit** if one of the prefixes of an observation(test run, i.e. trace from L^*_δ) is included in the observation objective.

Definition 2.4.1
Let $i \in \mathcal{IOTS}(L_I, L_U)$ be an implementation, $O \subseteq L^*_\delta$ a set of observations,

and $e \subseteq L_\delta^*$ a singular observation objective, then

1. $\mathbf{prefix}(O) = \{\sigma_1 \mid \sigma_1 \cdot \sigma_2 \in O \text{ with } \sigma_2 \in L_\delta^*\}$
2. i **rios** $e \ =_{\text{def}} \ \mathit{Straces}(i) \cap e \neq \emptyset$
3. $H_e^{\mathbf{rios}}(O) = \mathbf{hit} \ =_{\text{def}} \ \mathbf{prefix}(O) \cap e \neq \emptyset$

□

Plural observation objectives With a singular objective we can only require the exhibition of one trace from that observation objective. This limits the expressivity of an observation objective. We want to specify more than one trace that should be exhibited during the experiment. We can do this with *plural observation objectives*. A plural observation objective is composed out of multiple singular observation objectives, each of which should be individually exhibited during the execution of the test suite in order to satisfy the composed (plural) observation objective.

In **ioco** terms a plural observation objective is a set which is element of $TOBS = \mathcal{P}(\mathcal{P}(L_\delta^*))$ and the reveal relation is $\mathbf{rev} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{P}(\mathcal{P}(L_\delta^*))$. The exhibition of a plural test objective E requires in general more than one observation during testing, since seldom $\cap\{e \mid e \in E\} \neq \emptyset$, which can be satisfied by just one observation. We define the *reveal plural relation* **riop** and the **hit**-function $H_e^{\mathbf{riop}}(O)$, analogously to the singular case.

Definition 2.4.2
Let $i \in \mathcal{IOTS}(L_I, L_U)$ be an implementation, $O \subseteq L_\delta^*$ a set of observations, and $E \subseteq \mathcal{P}(L_\delta^*)$ a plural observation objective, then

1. i **riop** $E \ =_{\text{def}} \ \forall e \in E : i$ **rios** e
2. $H_E^{\mathbf{riop}}(O) = \mathbf{hit}$ iff $\forall e \in E : H_e^{\mathbf{rios}}(O) = \mathbf{hit}$

□

2.4.2 Test generation

We first consider the singular case. We have formal definitions of the **rev** relation and of the **imp** relation, resp. as **rios** and **ioco**. We want to generate a test suite that is sound and e-complete. Furthermore we restrict our observation objective such that $\{i \in \mathit{MODS} \mid i \ \mathbf{rios} \ e\} \cap \{i \in \mathit{MODS} \mid i \ \mathbf{ioco} \ s\} \neq \emptyset$ (c.f. Section 2.3 Equation 2.18). Observation objectives that are contained in the suspension traces of the specification (observation objective $e \subseteq \mathit{Straces}(s)$) satisfy this restriction. For such observation objectives Algorithm 5.6 in [dV01] derives a test suite that is sound and e-complete, by deriving a test case for each trace of the observation objective—note that e need not be finite, i.e. it may contain an infinite number of traces.

Algorithm 2.4.3
Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification, and let $e \subseteq \mathit{Straces}(s) \subseteq L_\delta^*$ be a singular observation objective. A test suite $T \subseteq \mathcal{TTS}(L_U, L_I)$ can be obtained by adding a test case $t_{\epsilon,\sigma}$ to T for every trace of e, i.e. $T = \{t_{\epsilon,\sigma} \mid \sigma \in e\}$. The test case $t_{\epsilon,\sigma}$ is obtained by application of the following rules:

1. (* terminate the test case when we reach the end of a guidance trace *)
   $t_{\rho,\epsilon} :=$ **pass**

2. (* the next item in the guidance trace is an input action a: apply it to the implementation, and be prepared to accept output, but then terminate the test case *)

$$\begin{aligned} t_{\rho,a\cdot\sigma'} := \quad & a; t_{\rho\cdot a,\sigma'} \text{ with } a \in L_I \\ + \quad & \sum\{x_i; \mathbf{pass} \mid x_i \in L_U \wedge x_i \in out(\, s \,\mathbf{after}\, \rho\,)\} \\ + \quad & \sum\{x_j; \mathbf{fail} \mid x_j \in L_U \wedge x_j \notin out(\, s \,\mathbf{after}\, \rho\,)\} \end{aligned}$$

3. (* the next item in the guidance trace is an output action x: check the next output of the implementation; terminate the test case when the output is not expected by model or guidance trace. (note that by our choice that observation objectives are $e \in \mathit{Straces}(s)$ it is the case that $x \in out(\, s \,\mathbf{after}\, \rho\,)\}$) *)

$$\begin{aligned} t_{\rho,x\cdot\sigma'} := \quad & x; t_{\rho\cdot x,\sigma'}, \text{ with } x \in L_U \cup \{\delta\} \\ + \quad & \sum\{x_i; \mathbf{pass} \mid x_i \in L_U \wedge x_i \in out(\, s \,\mathbf{after}\, \rho\,) \setminus \{x\}\} \\ + \quad & \sum\{x_j; \mathbf{fail} \mid x_j \in L_U \wedge x_j \notin out(\, s \,\mathbf{after}\, \rho\,)\} \end{aligned}$$

□

In order to judge if we have found evidence whether an implementation exhibits the observation objective, we collect all observations during testing, which gives us a set of test runs. We obtain the exhibition verdict by applying the **hit**-function $H_e^{\mathbf{rios}}$.

The test generation of test suites for plural observation objectives is straightforward. We flatten the set of singular observation objectives to one set and apply Algorithm 2.4.3. So, for specification $s \in \mathcal{LTS}(L_I, L_U)$ and plural observation objective $E \subseteq \mathcal{P}(\mathit{Straces}(s))$ a test suite $T \subseteq \mathcal{TTS}(L_U, L_I)$ can be obtained by applying Algorithm 2.4.3 with $e = \bigcup E$. We obtain the exhibition verdict by applying the **hit**-function $H_E^{\mathbf{riop}}$ for the set of test runs obtained during the execution of the test suite.

2.4.3 Including hit and miss verdicts into test cases

In the previous section we gave test derivation algorithm 2.4.3 that derives tests for **ioco** and **rios**, but where we have to apply the **hit**-function separately to obtain the exhibition verdict. With on line testing we restrict ourself to singular observation objectives, which can be exhibited by one observation of a test case execution (for plural observation objectives we need, by definition, multiple test runs). If such test case already contains the exhibition verdicts **hit** and **miss** (like it contains the verdicts **pass** and **fail**) we do not have to apply the **hit**-function separately. We adapt the **ioco** algorithm that we gave as Algorithm 2.2.18 on page 39 to support the use of guidance information and to include the exhibition verdicts into the derived test cases, and present the result as Algorithm 2.4.5 on the facing page.

The main difference between our adaptation of the **ioco** algorithm (Algorithm 2.4.5) and Algorithm 2.4.3 (that derives tests for **ioco** and **rios**) is that the latter shows how to derive a test case for a specific individual trace of the observation objective, where our algorithm essentially does the same random test derivation as in the **ioco** algorithm, with the random choices constrained to follow the observation objective.

Extended observation objectives In Algorithm 2.4.3 we considered observation objectives that are contained in the suspension traces of the specification. This means that a trace σ, that brings the IUT to a known error, (i.e. it makes the IUT respond with action x, where $\sigma{\cdot}x$ is not in the suspension traces of the specification) is among the observation objectives that we considered, but trace $\sigma{\cdot}x$ is not (it isn't a suspension trace of the specification).

However, we want to be able to use such trace $\sigma.x$ as observation objective: on an IUT that contains the error, a test case for it should yield verdict $\langle \mathbf{fail}, \mathbf{hit} \rangle$.

Therefore, for Algorithm 2.4.5 we consider a slightly larger class of observation objectives. Each trace of the observation objective either is *a suspension trace of the specification*, or it is a suspension trace of the specification *extended with one observation that is not in the specification.* Such latter trace is of the form $\sigma{\cdot}x$ with $\sigma \in \mathit{Straces}(s)$ and $x \in L_U^\delta$ and $\sigma{\cdot}x \notin \mathit{Straces}(s)$.

Function out_{tr} We use out_{tr} to denote the output actions (including δ) that are enabled in state p of a trace of an observation objective (treating δ as an ordinary output action, without synthesising it—this is where out_{tr} differs from out).

Definition 2.4.4
Let p be a state in a transition system, and P be a set of states, then

1. $out_{tr}(p) \ =_{\text{def}} \ \{x \in L_U^\delta \mid p \xrightarrow{x} \}$
2. $out_{tr}(P) \ =_{\text{def}} \ \bigcup\{ out_{tr}(p) \mid p \in P\}$

□

Algorithm 2.4.5
Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification, and let S be a non-empty set of states with initially $S = s \,\mathbf{after}\, \epsilon$. Let $g \in \mathcal{LTS}(L_I, L_U \cup \{\delta\})$ be a guidance specification, and let G be a non-empty set of states with initially $G = g \,\mathbf{after}\, \epsilon$. A test case $t \in \mathcal{TTS}(L_U, L_I)$ is obtained from S and G by a finite number of recursive applications of the following nondeterministic choices (rules). The resulting test case has a tree structure. Initially the tree consists of just the root node which acts as a placeholder to be expanded. Each rule expands a placeholder node, but may also add new placeholder nodes. In the rules we use the process-algebraic notation that we mentioned at the end of Section 2.2.1 to denote the construction of the tree. We use $\epsilon(p)$ to denote that at state p the end of the guidance trace is reached (i.e. p is a sink state). and $\epsilon(P)$ to denote that $\exists p \in P : \epsilon(p)$[1].

The rules are:

1a. (* for when we terminate the test case before we reach the end of a guidance trace; stop the recursion in the algorithm *)
$t := \langle \mathbf{pass}, \mathbf{miss} \rangle$ if $\not\epsilon(G)$

This rule turns a placeholder t into an end state $\langle \mathbf{pass}, \mathbf{miss} \rangle$.

[1] In the implementation in TorX and JTorX we typically mark the end of a guidance trace in an observation objective LTS with a self-loop with label ϵ.

1b. (* terminate the test case when we reached the end of a guidance trace; stop the recursion in the algorithm *)
$t := \langle \mathbf{pass}, \mathbf{hit} \rangle$ if $\epsilon(G)$

This rule turns a placeholder t into a sink state $\langle \mathbf{pass}, \mathbf{hit} \rangle$.

2. (* give a next input to the implementation, and be prepared to accept output *)

$$
\begin{aligned}
t := \quad & a; t_a \\
+ \quad \sum \quad & \{x_j; \langle \mathbf{fail}, \mathbf{miss} \rangle \mid x_j \in L_U \wedge x_j \notin out(S) \\
& \wedge (x_j \notin out_{tr}(G) \vee (x_j \in out_{tr}(G) \wedge \not\epsilon(\, G \,\mathbf{after}\, x_j\,)))\} \\
+ \quad \sum \quad & \{x_j; \langle \mathbf{fail}, \mathbf{hit} \rangle \mid x_j \in L_U \wedge x_j \notin out(S) \\
& \wedge (x_j \in out_{tr}(G) \wedge \epsilon(\, G \,\mathbf{after}\, x_j\,))\} \\
+ \quad \sum \quad & \{x_i; \langle \mathbf{pass}, \mathbf{miss} \rangle \mid x_i \in L_U \wedge x_i \in out(S) \\
& \wedge x_i \notin out_{tr}(G)\} \\
+ \quad \sum \quad & \{x_i; \langle \mathbf{pass}, \mathbf{hit} \rangle \mid x_i \in L_U \wedge x_i \in out(S) \\
& \wedge x_i \in out_{tr}(G) \wedge \epsilon(\, G \,\mathbf{after}\, x_i\,)\} \\
+ \quad \sum \quad & \{x_i; t_{x_i} \mid x_i \in L_U \wedge x_i \in out(S) \\
& \wedge x_i \in out_{tr}(G) \wedge \not\epsilon(\, G \,\mathbf{after}\, x_i\,)\}
\end{aligned}
$$

where $a \in L_I$, $S \,\mathbf{after}\, a \neq \emptyset$ (i.e. $a \in in(S)$) and $G \,\mathbf{after}\, a \neq \emptyset$ (i.e. $a \in in(G)$),
and t_a is $\langle \mathbf{pass}, \mathbf{hit} \rangle$ if $\epsilon(\, G \,\mathbf{after}\, a\,)$,
and otherwise ($\not\epsilon(\, G \,\mathbf{after}\, a\,)$) t_a is obtained by recursively applying the algorithm on $S \,\mathbf{after}\, a$ and $G \,\mathbf{after}\, a$
and for each $x_i \in out(S)$, t_{x_i} is obtained by recursively applying the algorithm for the set of states $S \,\mathbf{after}\, x_i$ and $G \,\mathbf{after}\, x_i$.

With $t := a; t'$ we mean that we add a state t' and a transition $t \xrightarrow{a} t'$.

This rule extends the tree by turning placeholder t into a non-leaf node by creating from it a whole set of edges: one for the input that we want to give, and one for each output label (but not for quiescence).

The edge for the input is labelled with an input action a, arbitrarily chosen from those input actions enabled at this point in the specification that are also enabled in the guidance information. This edge points to a newly created node, which is either a newly created placeholder t_a, when the end of the guidance trace is not reached, or sink state $\langle \mathbf{pass}, \mathbf{hit} \rangle$, when the end of the guidance trace is reached.

Those edges that correspond to output actions that are not present in (expected by) the specification at that point lead to a sink state that contains **fail**. If the output action is not present in the guidance information at that point, or not the last action of the guidance trace, the sink state also contains **miss**; otherwise it contains **hit**.

Those edges that correspond to output actions that are present in (expected by) the specification at that point, but are not present in the guidance information at that point, lead to a sink state that contains $\langle \mathbf{pass}, \mathbf{miss} \rangle$.

Those edges that correspond to output actions that are present in the specification and are present in the guidance information and are the last

action of a guidance information trace lead to a sink state that contains $\langle \mathbf{pass}, \mathbf{hit} \rangle$.

Those edges that correspond to output actions that are present in the specification and are present in the guidance information but are not the last action of a guidance information trace lead to newly created placeholders.

3. (* check the next output of the implementation *)

$$
\begin{aligned}
t := \quad & \sum \quad \{x_j; \langle \mathbf{fail}, \mathbf{miss} \rangle \mid x_j \in (L_U^\delta) \wedge \ x_j \notin out(S) \\
& \qquad \wedge (x_j \notin out_{tr}(G) \vee (x_j \in out_{tr}(G) \wedge \not\epsilon(\, G \ \mathbf{after}\ x_j\,)))\} \\
+ \quad & \sum \quad \{x_j; \langle \mathbf{fail}, \mathbf{hit} \rangle \mid x_j \in (L_U^\delta) \wedge x_j \notin out(S) \\
& \qquad \wedge (x_j \in out_{tr}(G) \wedge \epsilon(\, G \ \mathbf{after}\ x_j\,)))\} \\
+ \quad & \sum \quad \{x_i; \langle \mathbf{pass}, \mathbf{miss} \rangle \mid x_i \in (L_U^\delta) \wedge x_i \in out(S) \wedge x_i \notin out_{tr}(G)\} \\
+ \quad & \sum \quad \{x_i; \langle \mathbf{pass}, \mathbf{hit} \rangle \mid x_i \in (L_U^\delta) \wedge x_i \in out(S) \\
& \qquad \wedge x_i \in out_{tr}(G) \wedge \epsilon(\, G \ \mathbf{after}\ x_i\,)\} \\
+ \quad & \sum \quad \{x_i; t_{x_i} \mid x_i \in (L_U^\delta) \wedge x_i \in out(S) \\
& \qquad \wedge x_i \in out_{tr}(G) \wedge \not\epsilon(\, G \ \mathbf{after}\ x_i\,)\}
\end{aligned}
$$

where t_{x_i} is obtained by recursively applying the algorithm on $S \ \mathbf{after}\ x_i$ and $G \ \mathbf{after}\ x_i$.

This rule extends the tree by turning a placeholder t into a non-leaf node by creating from it a whole set of edges, one for each output label (including quiescence). This rule is very similar to the part of rule 2 that describes dealing with accepted output: the only difference is that we accept quiescence here, and we do not accept it (wait for it to be "observed") in rule 2.

□

In the algorithm we not only have rules 1a and 1b to end the test case with a verdict that contains **pass**, but we also in rules 2 and 3 explicitly test if we reached the end of a guidance trace. Rule 1a is present for generality, to allow user interaction during the guided testing: it allows us to give the right verdict when a user stops testing prematurely (when left alone, the algorithm will never choose rule 1a). Rule 1b is present only for the case where initially already $\epsilon(G)$; in rules 2 and 3, we test whether we reached the end of the guidance trace when we might recursively apply the algorithm.

Comparison with Algorithm 2.2.18 We now compare the guided algorithm with the non-guided one.

- Both algorithms have a similar structure of (essentially) three rules.
- In both algorithms we can always replace a placeholder by a sink state with label **pass** (rule 1).
- In both algorithms we replace a placeholder by an edge leading to a sink state with label **fail** (rules 2 and 3) under the same condition on the specification: $x_j \in L_U \wedge x_j \notin out(S)$ (rule 2) resp. $x_j \in (L_U^\delta) \wedge x_j \notin out(S)$ (rule 3).

- In the guided algorithm there are more cases in rules 2 and 3 where we replace a placeholder by an edge leading to a sink state with label **pass** than in the non-guided one: these are the cases where we can decide that we **hit** or **miss** an observation objective. In the non-guided algorithm we would create a placeholder instead of the sink state. However, because we can always replace a placeholder by a sink state with label **pass**, we can with the non-guided algorithm obtain a test case with the same sink states (only looking at **pass** and **fail** verdicts) as with the guided one.

2.4.4 Conclusion

To conclude our discussion of how we instantiate the framework we give in Table 2.4 an updated version of Table 2.3 on page 46, in which the instantiations that we have chosen are shown.

Bibliographical note This section, up-to (but not including) Section 2.4.3, has been taken from [dV01]. Algorithm 2.4.3 in Section 2.4.2 is based on Algorithm 5.6 of [dV01]; we extended it to generate input-enabled test cases. Section 2.4.3 is our own extension of the existing work.

2.5 Summary

We have presented the framework of formal methods in conformance testing, and instantiated it with implementation relation **ioco**. For **ioco** sound test cases can be produced, resulting in an exhaustive test suite "in the limit" of running the test derivation algorithm. We discussed an extension to guide the test derivation. We briefly mentioned a few generalisations and variants of **ioco**: implementation relations $\mathbf{ioco}_{\mathcal{F}}$, $\mathbf{mioco}_{\mathcal{F}}$, **uioco**, $\mathbf{tioco}_{M}$, **rtioco** and **sioco**.

physical ingredients:	
black-box implementation	IUT can be any computer system or program which can can be modelled as input-output transition system, where inputs and outputs can be mapped one-to-one on L_I resp. L_U and inputs can always occur
execution of a test case	$\mathrm{EXEC}(t, \mathrm{IUT})$: physical execution of test case t on IUT; may involve multiple runs, e.g. to cater for non-determinism
formal ingredients:	
specification	$s \in \mathcal{LTS}(L_I, L_U)$
model of implementation	$i_{\mathrm{IUT}} \in \mathcal{IOTS}(L_I, L_U)$
implementation relation	**ioco**
observation objective	$e \in \mathcal{P}(L_\delta^*)$
reveal relations	**rios** (singular o.o.), **riop** (plural o.o.)
test case	$t \in \mathcal{TTS}(L_U, L_I)$
observations	$(L \cup \{\delta\})^*$
model of execution of a test case	$obs : TESTS \times MODS \rightarrow \mathcal{P}(OBS)$
verdict functions	$v_t : \mathcal{P}((L \cup \{\delta\})^*) \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$
hit functions	$H_e^{\mathbf{rios}} : \mathcal{P}((L \cup \{\delta\})^*) \rightarrow \{\mathbf{hit}, \mathbf{miss}\}$
hit functions	$H_E^{\mathbf{riop}} : \mathcal{P}((L \cup \{\delta\})^*) \rightarrow \{\mathbf{hit}, \mathbf{miss}\}$
test derivation algorithm	$gen_{\mathbf{ioco},\mathbf{rios}} : \mathcal{LTS}(L_I, L_U) \rightarrow \mathcal{P}(\mathcal{TTS}(L_U, L_I))$
assumptions:	
test hypothesis	for all IUT some i_{IUT} models IUT $obs(t, i_{\mathrm{IUT}})$ models $\mathrm{EXEC}(t, \mathrm{IUT})$
prove obligations:	
soundness	$gen_{\mathbf{ioco},\mathbf{rios}}$ is sound for any $s \in \mathcal{LTS}(L_I, L_U)$
exhaustiveness	$gen_{\mathbf{ioco},\mathbf{rios}}$ is exhaustive for any $s \in \mathcal{LTS}(L_I, L_U)$
e-soundness	$gen_{\mathbf{ioco},\mathbf{rios}}$ is e-sound for any $e \in TOBS$ for which $\{i \mid i \ \mathbf{rev}\ e\} \cap \{i \mid i \ \mathbf{imp}\ s\} \neq \emptyset$
e-exhaustiveness	$gen_{\mathbf{ioco},\mathbf{rios}}$ is e-exhaustive for any $e \in TOBS$ for which $\{i \mid i \ \mathbf{rev}\ e\} \cap \{i \mid i \ \mathbf{imp}\ s\} \neq \emptyset$

Table 2.4: Our instantiation of the framework for conformance and exhibition testing

Chapter 3

Architecture of TorX

In the previous chapters we introduced our design goal and presented the theory that we build on. In this chapter we present the global architecture of our design; we refer to the design as TorX. In Chapter A we discuss two implementations of this design: the model-based testing tools TorX and JTorX.

We start this chapter with a very abstract view of the architecture in Section 3.1. In Section 3.2 we make the connection between this abstract view and the theory that we discussed in Chapter 2.

In the remainder of this chapter, and in the following chapters, we refine this abstract architectural view to fulfil our design goal, where we look in particular at requirement 3 (the design should be suitable for on-line and off-line testing) and requirement 7 (the design should support random and guided on-line testing). We do this by discussing configurations of TorX that correspond with these requirements, as shown in Table 3.1.

		random	guided
on-line		Sect. 3.4	Sect. 3.5
off-line (Sect. 3.6)	exhaustive	Sect. 3.6.3	Sect. 3.6.5
	non-exhaustive	Sect. 3.6.4	Sect. 3.6.5

Table 3.1: Overview of TorX configurations discussed, with section numbers.

For each configuration we show its decomposition in components, and the interfaces between them, and we show how those components that "are in charge" can accomplish their task using the other components. We use a running example to illustrate each scenario; we introduce the example in Section 3.3.

In Section 3.4 we discuss on-line testing with random test selection, which we in Section 3.5 extend with guided test selection (i.e. the use of test purposes to guide test derivation). In Section 3.6 we discuss off-line testing with systematic exhaustive test derivation, derivation with random test selection, and guided derivation of test cases. Section 3.7 ends the chapter with a summary of the inter-component interfaces.

Typographic conventions In the previous chapter we typically used *a slanted font* for 'mathematical' functions, e.g. those on LTSes, like *in* and *out*. In this chapter we introduce interface functions for which we use a sans serif font. Some of the functions that we introduce here have names that coincide with names of 'mathematical' functions of the previous chapter, e.g. in and out, but which thus can be distinguished by font.

3.1 Starting Point

We start with the most abstract view of the TorX architecture and its environment, depicted in Figure 3.1. This view is obtained by taking the abstract view of testing that we gave in Chapter 1, 'Introduction' as Figure 1.1 on page 2, and extending it with the model (Specification), to reflect the fact that we are dealing with model-based testing.

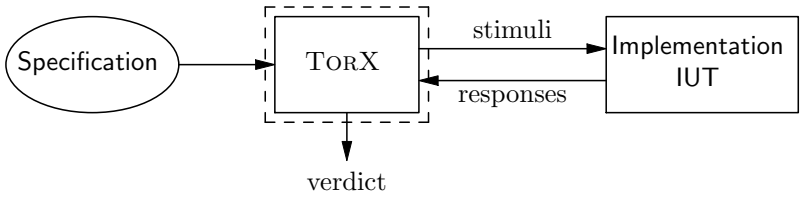


Figure 3.1: Abstract view of TorX: it interacts with Implementation via stimuli and responses. It derives the stimuli and predictions for the responses from Specification. Testing results in a verdict.

In Figure 3.1 the tester (TorX) itself is shown as a black box. We surrounded the tester (TorX) with a dashed box to indicate the border between the tester and its environment. When we decompose the tester in subsequent figures, we also show the dashed box there. The environment of TorX consists of the Specification (at its left side in the figure), and the Implementation (at its right side in the figure). Neither Specification nor Implementation are part of TorX itself, but TorX uses the Specification (to derive tests from) and interacts with the Implementation (by "playing the role" of its environment).

Note that we have omitted the user from the picture. For now we only talk about automated testing for which we do not need a user. We come back to the user later, in Section 3.5.

3.2 Link with Theory

In the previous chapter we have seen the formal framework of model-based testing, and the instantiation of it for the implementation relation **ioco**.

Formal framework The formal framework was schematically presented in Figure 2.1 on page 22. From the formal model we take the basic testing concepts that we use in the architecture: specification, test purpose (a.k.a. ob-

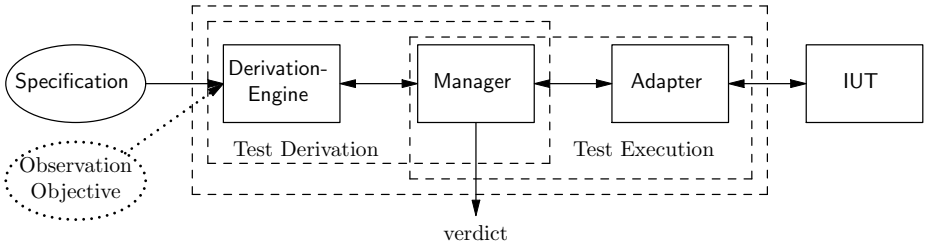


Figure 3.2: Schematic view of *on-line* model-based testing. Test derivation (top inner box) and test execution (bottom inner box) take place in an integrated manner. Each test step that is derived from Specification, and optional Observation Objective, is immediately executed on the IUT, and only then the next test step is derived. Thus, a test case is only implicitly present, and therefore there is no test suite in this figure. Execution of the test on IUT yields a verdict.

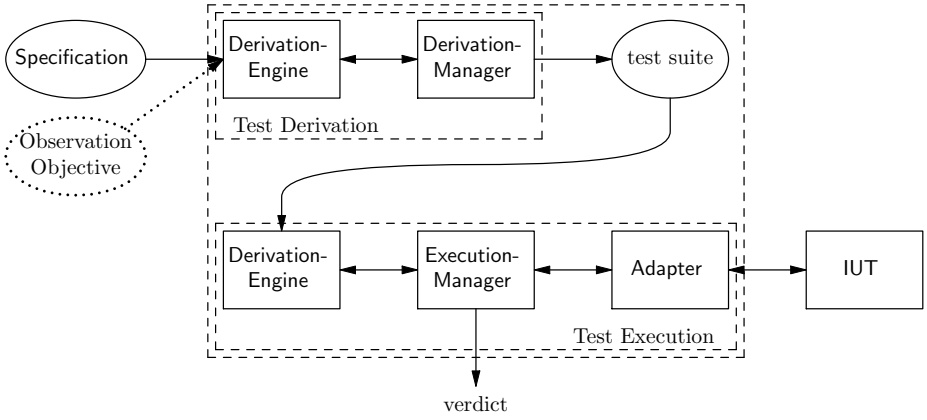


Figure 3.3: Schematic view of *off-line* model-based testing. Test derivation, in the top inner box, uses Specification, and optional Observation Objective, as input and produces a test suite. Test execution, in the bottom inner box, applies the test suite to the IUT, to yield a verdict.

servation objective), test derivation, test suite, implementation, test execution, and verdict. We leave out the implementation relation because its role in the architecture is only indirect (it is implicitly present in the test derivation). We rearrange these elements of Figure 2.1 into a form that is more suitable for use as basis of the architecture for on-line (Fig. 3.2) and off-line (Fig. 3.3) testing. In both these figures we already "opened up" the test derivation and test execution boxes; the outer dashed box separates the components of the architecture from their environment.

In Figure 3.2 we show our basic architecture for on-line testing. Here, no test suite is shown, because tests are only implicitly present: recall that each test step that is derived, is immediately executed. The integration between test

derivation and test execution is shown: they share the Manager component.

Figure 3.3 targets off-line testing, as the presence of the test suite indicates. This figure has the same components as the abstract view of off-line testing of Figure 1.3 on page 4, except for the model that had not yet been introduced at that point.

The formal framework and its instantiation with **ioco** leave choices open, like about the relation between test derivation and test execution (on-line or off-line?) and about how to do test selection. When a tool that is based on this theory is used to do actual testing, all these open choices will have to be resolved. For TorX we resolved some of them by our design choices, discussed in Chapter 1; others we made part of the design goal, and thus they are passed on to the user as user-configurable options of TorX.

Test Derivation As laid down by requirements 1 and 2 we use the **ioco** algorithm (discussed in Section 2.2.5) to derive test cases from a labelled transition system (LTS) representation of the specification. How we obtain an LTS from a specification in an "arbitrary" formalism is outside the scope of the **ioco** theory, and, as laid down by requirement 5, we strive for language independence. Nevertheless, obtaining the LTS from a given model is an indispensable part of a working testing tool. For now, we assume that the test derivation tool component will take care of this; we will discuss this in greater detail in Chapter 4.

As laid down in requirement 3, the tool should be suitable for both on- and off-line testing. To obtain a test *suite* (multiple different test cases) we have to run the **ioco** algorithm multiple times and vary the two choices that the **ioco** algorithm leaves open: the choice of the next test step (do we try to stimulate, do we observe, or do we stop?) and the choice of the particular stimulus when we have chosen to stimulate. This will give us an exhaustive test suite "in the limit".

Test Execution In the formal framework, test execution function EXEC takes a test suite and an IUT and returns a verdict. In the **ioco** testing theory with which we instantiated the formal framework, the execution of a test suite consists of the application of each of its test cases to the IUT. The test cases themselves are (by construction) deterministic, but when there is non-deterministic behaviour in the system under test, or when the system under test contains uncontrollable choices (such that the same sequence of stimuli yields different responses) the repeated application of a test case may lead to different observations. How often test cases must be applied and in which order are choices left open by the formal framework and the **ioco** testing theory. The implementations of our design leave this to their user.

3.3 Running Example

We will illustrate each of the configurations of Table 3.1 with a small example. For this example we will use a model, depicted in Figure 3.4, that is based on

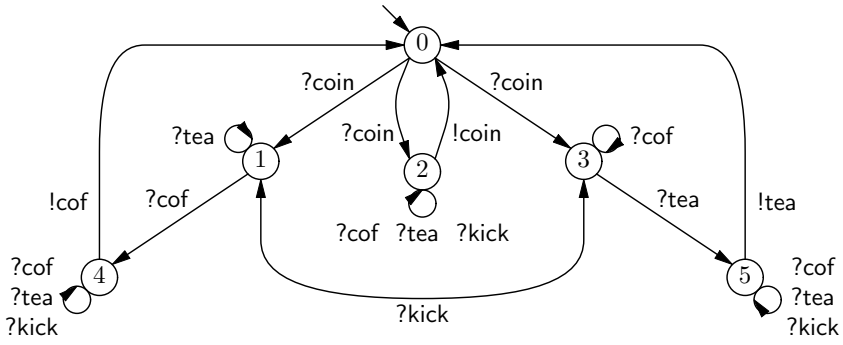


Figure 3.4: Quirky Coffee Machine (Specification)

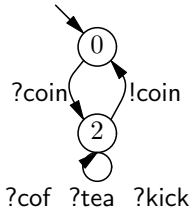


Figure 3.5: Refund-only Machine (not input-enabled; when used as implementation, missing or omitted inputs are assumed to be on self-loops, see the text).

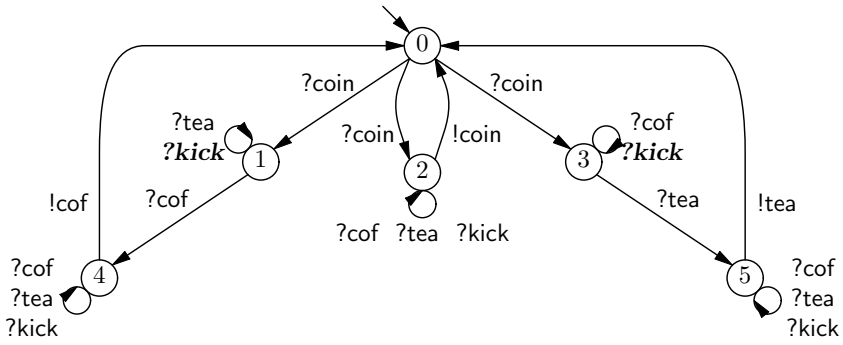


Figure 3.6: Kick-insensitive Machine: ?kick in states 1 and 3 has no effect. (not input-enabled; when used as implementation, missing or omitted inputs are assumed to be on self-loops, see the text).

the *Quirky Coffee machine* of Rom Langerak [Lan90]. In this model we prefix inputs (to the system under test) with a question mark (?) and outputs with an exclamation mark (!). The model describes a machine that is able to produce coffee and tea, although obtaining the preferred beverage may take some effort. When a coin is inserted the machine nondeterministically goes to one of three states: 1, 2, or 3. In state 2 it will immediately return the coin and return to the initial state. In state 1 only the functionality to produce coffee, and in state 3 only the functionality to produce tea is working. In states 1 and 3, when the button is pressed for the beverage that cannot be produced, the machine does nothing (modelled as a self loop). However, kicking the machine, while it is in one of these states, makes it change state (from 1 to 3, and vice versa), such that the functionality for the other beverage (and only for that one) is working. So, when the machine consumes an inserted coin (and does not immediately return it), but does not produce coffee when the coffee button is pressed, it is possible to obtain coffee by kicking the machine and pressing the coffee button once more. Tea can be obtained in a similar way.

Implementations In the examples we use two different implementations of this machine: a refund-only one, and a kick-insensitive one. Note that the implementation examples that we present here do not allow all inputs in all states, i.e. they are not input-enabled. We assume that any model, which is used as implementation, is (implicitly) made input-enabled by adding self-loops for the missing inputs—our tool implementation (i.e. the Adapter for use of simulated model as IUT, design req. 15) does such *input-completion* automatically. This allows us to use also models, that are not input-enabled, as implementation.

The refund-only implementation, depicted in Figure 3.5, never serves coffee or tea, but only refunds money. It is **ioco**-correct w.r.t. the specification.

The kick-insensitive implementation, depicted in Figure 3.6, does serve coffee and tea, but deviates from the specification in its behaviour when it is kicked. In the specification, a kick in the right states (state 1 or 3) triggers a transition to a new state, in which the beverage is available that before was inaccessible. However, in the kick-insensitive implementation, kicking does not help: the machine remains in the same state, and thus, kicking does not affect which beverage can be served. We illustrate this by showing the sets of expected outputs for the traces that describe these scenarios, where we use i for the kick-insensitive implementation, and s for the specification.

with $\sigma_2 = ?\mathsf{coin} \cdot ?\mathsf{cof} \cdot \delta \cdot ?\mathsf{kick} \cdot ?\mathsf{cof}$:
$$out(\, i \,\mathbf{after}\, \sigma_2 \,) = \{\delta\} \not\subseteq \{!\mathsf{cof}\} = out(\, s \,\mathbf{after}\, \sigma_2 \,)$$
with $\sigma_1 = ?\mathsf{coin} \cdot ?\mathsf{tea} \cdot \delta \cdot ?\mathsf{kick} \cdot ?\mathsf{tea}$:
$$out(\, i \,\mathbf{after}\, \sigma_1 \,) = \{\delta\} \not\subseteq \{!\mathsf{tea}\} = out(\, s \,\mathbf{after}\, \sigma_1 \,)$$

Moreover, the behaviour of the kick-insensitive implementation also differs from the specification when, after not receiving the requested beverage and then kicking the machine, one asks for the *opposite* beverage: the kick-insensitive implementation serves it, but the specification does not (and remains quiescent).

with $\sigma_3 = ?\mathsf{coin} \cdot ?\mathsf{cof} \cdot \delta \cdot ?\mathsf{kick} \cdot ?\mathsf{tea}$:
$$out(\, i \,\mathbf{after}\, \sigma_3 \,) = \{!\mathsf{tea}\} \not\subseteq \{\delta\} = out(\, s \,\mathbf{after}\, \sigma_3 \,)$$

with $\sigma_4 = ?\mathsf{coin} \cdot ?\mathsf{tea} \cdot \delta \cdot ?\mathsf{kick} \cdot ?\mathsf{cof}$:

$$out(\, i \,\mathbf{after}\, \sigma_4 \,) = \{!\mathsf{cof}\} \not\subseteq \{\delta\} = out(\, s \,\mathbf{after}\, \sigma_4 \,)$$

This shows that the kick-insensitive implementation is **ioco**-*in*correct w.r.t. the specification.

3.4 Random On-Line Testing

When we integrate test derivation and test execution to the point where each test step is derived "on demand" driven by the test execution, i.e. when we do on-line testing, we can simplify the architecture that we showed in Figure 2.1.

This section is structured as follows. After defining components (Section 3.4.1) and interfaces (Section 3.4.2), we show how these can be used for random on-line testing (Section 3.4.3), and illustrate this using our example (Section 3.4.4).

3.4.1 Components

In our schematic view of model-based testing of Figure 2.1 we have separate activities Test Derivation and Test Execution, with the test suite as object that is passed from the former to the latter. Conceptually, we use the approach of separation of concerns to decompose both test derivation and test execution in components that provide access to the Specification (the DerivationEngine component) resp. the IUT (the Adapter component), and a Manager component that is responsible for the progress, and for resolving any remaining open choices in test derivation resp. test execution. Figure 3.2, without the optional observation objective, depicts the architecture. (We discuss the case where the observation objective is present in Section 3.5.)

The DerivationEngine hides the details of the Specification from the Manager. The DerivationEngine computes the information (those "primitives") that the Manager needs from the Specification. As we will see, the DerivationEngine provides access to the suspension automaton of Specification; the Manager uses this to implement the test derivation algorithm. In this way, the Manager is as independent as possible from the Specification; in particular, it is independent from its syntactical format, and the Manager only has to resolve the open choices in the test derivation algorithm: it decides which test case to construct.

The Adapter hides IUT-specific details from the Manager (which is IUT-independent) and provides it with a uniform interface to the IUT. The Adapter is, by definition, IUT-dependent. It provides the connection to the IUT, which encompasses the connection proper as well as the mapping between the *abstract* labels of the Specification (and thus of the test case) and the *concrete* interactions with the IUT. This mapping in the Adapter resolves the remaining open choices for test execution: the choice of interaction details (like test data) that are not represented in (have been abstracted away from) the stimuli labels.

For on-line testing we don't have to pass a test suite from test derivation to test execution – passing one test step at a time suffices, and thus we omit the test suite from the architecture for on-line testing.

3.4.2 Interfaces

Here we discuss the interfaces between the components in our decomposition. Before we look at the individual interfaces we discuss the form of our interfaces in general.

In general our interfaces are between two components, and in general the interaction between them follows a question-answer (request-response, client-server) scheme: one of them always takes the initiative, the other always responds. Components may exchange formal objects over the interfaces, like labels, states or sets of these. For now we do not care about the specific way in which the interfaces may be implemented, nor about the concrete representation of the formal objects that are exchanged over them. We therefore present the interfaces as functions of which we give the signature, typically in terms of labels, states, or sets of these. For the discussion in this thesis, we assume that no errors result from the application of the interface functions, and thus they do not appear in the interface definitions.

In general we strive for lean interfaces. Lean interfaces between modules are an indication of high independence between the modules, which is a sign of successful decomposition. Two practical consequences of lean interfaces are runtime efficiency (reduced amount of data to be passed between the components) and implementation efficiency (reduced number of interface functions that have to be implemented).

Between Manager and Adapter

Inspiration for the interface between Manager and Adapter comes from the testing process. When the test is derived and executed one step at a time, these test steps, i.e. the stimuli that are to be given and the observed responses, are natural elements in the interface between Manager and Adapter.

Because the testing is "driven" by the Manager, we choose that the Manager will take the initiative on this interface. The interface must at least be capable of transferring a stimulus from Manager to Adapter, and transferring an observation in the opposite direction. As mentioned before, we use labels from $L_I \cup L_U^\delta$ to represent stimuli and observations. To this we add functions for initialisation and cleanup. Our interface thus exists of the following four functions, to be implemented by the Adapter. (We explain below why tryStim and getObs return a tuple instead of just a label.)

1. start $:\rightarrow$ void
2. tryStim : $L_I \rightarrow \{\text{I}, \text{U}\} \times (L_I \cup L_U)$
3. getObs $:\rightarrow \{\text{U}\} \times (L_U \cup \{\delta\})$
4. stop $:\rightarrow$ void

Table 3.2: Signature of Adapter interface functions.

Ad 1: start With function start the Manager can ask the Adapter to start the IUT (or make a connection to it, or reserve resources, or anything else that is necessary for a specific IUT).

Ad 2: tryStim With function tryStim the Manager can ask the Adapter to attempt to apply a given stimulus. By construction, our test cases are able to consume observations that are (turn out to be) pending when an attempt is made to apply a stimulus (see property 4 of Definitions 2.2.13 on page 34, and step 2 of Algorithm 2.2.18). We delegate the issue of maintaining a queue of pending observations to the Adapter, and provide minimal (but sufficient) support via our definition of tryStim. It is defined as follows.

$$\mathsf{tryStim}(a) \;=_{\text{def}}\; \begin{cases} \langle \text{U}, x \rangle \text{ with } x \in L_U & \text{if } x \text{ was produced before } a \text{ could be applied} \\ \langle \text{I}, a \rangle \text{ with } a \in L_I & \text{otherwise (i.e. } a \text{ was applied to the IUT)} \end{cases}$$

The return value of tryStim is a tuple $\langle t, l \rangle$. It contains a type $t \in \{\text{I}, \text{U}\}$ and a label l. The type t is there to be able to distinguish the two kinds of return values without knowledge of the label sets L_I and L_U. The label l is either

- an acknowledgement of the successful application of stimulus a (when there was no pending observation), or
- the first pending observation, x (when there was a pending observation).

The acknowledgement consists of the stimulus that was applied, because then in either case the Adapter returns a representation of the latest interaction between Adapter and IUT. Application of a stimulus should always succeed when there is no observation pending, because the IUT is assumed to be input-enabled.

Ad 3: getObs With function getObs the Manager can ask the Adapter for an observation. It is defined as follows.

$$\mathsf{getObs}() \;=_{\text{def}}\; \begin{cases} \langle \text{U}, x \rangle \text{ with } x \in L_U & \text{when } x \text{ was produced by the IUT} \\ \langle \text{U}, \delta \rangle & \text{when the IUT did not produce output} \end{cases}$$

For consistency with tryStim also getObs returns a tuple $\langle t, l \rangle$. Here t is always U, and l is either the first pending or freshly observed observation, or δ (as representation of quiescence).

Ad 4: stop With function stop the Manager can ask the Adapter to stop the IUT (or break the connection to it, or release resources, or anything else that is necessary for a specific IUT).

Between Manager and DerivationEngine

The Manager drives the test derivation algorithm and resolves its open choices, and uses the DerivationEngine to access the specification. In the interaction between Manager and DerivationEngine it always is the former that takes the initiative.

1. $\mathsf{start} :\rightarrow P$
2. $\mathsf{in} : P \rightarrow \mathcal{P}(L_I)$
3. $\mathsf{out} : P \rightarrow \mathcal{P}(L_U \cup \{\delta\})$
4. $\mathsf{next} : P \times (L_I \cup L_U \cup \{\delta\}) \rightarrow P \uplus \{\bot\}$

Table 3.3: Signature of DerivationEngine interface functions (non-guided), given in terms of label(set)s L_I, L_U and δ, and the pseudo-state type of Table 3.4.

5. $\mathsf{PS} : S_\Gamma \rightarrow P$
6. $\mathsf{m} : P \rightarrow S_\Gamma$

Table 3.4: Signature of pseudo-state type (used in non-guided DerivationEngine interface).

Our interface consists of the four functions, to be implemented by the DerivationEngine, given in Table 3.3.

Inspiration for the interface between Manager and DerivationEngine comes from the **ioco** test derivation Algorithm 2.2.18. In each step of the algorithm, the next step in the test case is derived from a set of states from the specification which we will call S in the following. In particular, each instance of S encountered during test derivation corresponds to a state of Γ_s, the suspension automaton of s (defined in Definition 2.2.12). At the start of the algorithm this set is s_0 **after** ϵ. To derive a test step the algorithm looks at the sets of stimuli and responses that are enabled in S, i.e. at $in(S)$ resp. $out(S)$. For the next step the algorithm looks at S **after** l, the states that can be reached from S via a transition with observable label l. We thus let the DerivationEngine provide access to the suspension automaton states, and to the sets of stimuli and responses that are enabled in these states.

For now we do not care how the suspension automaton states are represented on the interface (whether as actual sets of state representations in serialised form, as set of state identifiers, or as state set identifiers) nor where the actual states and state sets are stored (in DerivationEngine or Manager– when states are stored in DerivationEngine, the interface needs an additional function to allow the Manager to indicate which states or state sets can be deleted). We make this explicit, by using an indirection: functions start and next return a *pseudo-state* object, and functions in, out and next accept a pseudo-state as argument. The signature of the pseudo-state functions is given in Table 3.4. We let PS construct a pseudo-state from a suspension automaton state, and let method m return the suspension automaton state from a pseudo-state:

$$\mathsf{PS}(s).\mathsf{m}() \ =_{\text{def}} \ s$$

Below we give the functionality of the DerivationEngine interface functions in terms of the functions *in*, *out* and **after** of Chapter 2, and the pseudo-state functions of Table 3.4. (In Chapter 4 we discuss how the interface functions can be implemented). There s is the (initial state of) the specification, and P is a pseudo-state (that gives access to a suspension automaton state, i.e. to a set of

states).

$$
\begin{aligned}
\mathsf{start}() &=_{\text{def}} \mathsf{PS}(\, s \,\mathbf{after}\, \epsilon \,) \\
\mathsf{in}(P) &=_{\text{def}} in(P.\mathsf{m}()) \\
\mathsf{out}(P) &=_{\text{def}} out(P.\mathsf{m}()) \\
\mathsf{next}(P, l) &=_{\text{def}} \begin{cases} \bot & \text{if } P.\mathsf{m}() \,\mathbf{after}\, l = \emptyset \\ \mathsf{PS}(\, P.\mathsf{m}() \,\mathbf{after}\, l \,) & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that we let $\mathsf{next}(P, l)$ return the special value $\bot$ when there is no outgoing transition from $P.\mathsf{m}()$ that has observable label l. We used special value $\bot$ to make it stand out. Moreover, $\emptyset$, i.e. the empty set of states of the specification, is not part of S_Γ, as given in Definition 2.2.12—otherwise we could have used $\mathsf{PS}(\emptyset)$ instead of $\bot$.

3.4.3 Manager Algorithm

Above we mentioned that when we do on-line derivation and execution we do not explicitly use a test suite, nor even a test case. We show this in Algorithm 3.1 for random on-line test derivation and execution, presented on the following page. Algorithm 3.1 is based on Algorithm 2.2.18 on page 39, and we discuss the correspondence below.

Algorithm 2.2.18 constructs a test case by the application of three rules:

1. to terminate the test case at any point with a **pass** verdict,
2. to try to apply a stimulus (and be ready to consume and check an observation), and
3. to obtain and check an observation.

Algorithm 2.2.18 keeps a state set S, with initial value $s \,\mathbf{after}\, \epsilon$ as its main data structure.

Algorithm 3.1 uses a pseudo-state P, that gives access to the same state set S, with the same initial value (set at line 2). It applies the first rule at line 5 when a given maximal length for the test run (*intended depth*) has been reached. It bases its choice between the second and the third rule on the knowledge whether stimuli are enabled, and on knowledge whether the previous interaction consisted of the observation of quiescence, as follows (see lines 7–12, and lines 4 and 25).

- If no stimuli are enabled, it applies the third rule;
- otherwise, if the previous interaction was an observation of quiescence, it applies the second rule (this is an heuristic that tries to avoid consecutive observations of quiescence);
- otherwise it randomly chooses one of the rules.

The application of the second rule starts at line 14, and the application of the third rule at line 17. For both rules, l, the result of the interaction with the IUT, is checked by asking the DerivationEngine to compute the successor pseudo-state P' (at line 19), and evaluating whether $P' = \bot$ at line 20. (By the definition of next, P' will be either $\bot$ or $\mathsf{PS}(\, P.\mathsf{m}() \,\mathbf{after}\, l \,)$.)

Algorithm 3.1: Random On-Line Test Derivation and Execution

```
   input  : A DerivationEngine d that gives access to the Specification and
            an Adapter a that gives access to the SUT
   output: A verdict
 1 begin
 2     P ⟵ d.start()
 3     a.start()
 4     prevLabel ⟵ ⊥
 5     while intended depth not reached do
 6         stimuli ⟵ d.in(P)
 7         if stimuli = ∅ then
 8             action ⟵ observe
 9         else if prevLabel = δ then
10             action ⟵ stimulate
11         else
12             action ⟵ random choice from {stimulate, observe}
13         switch action do
14             case stimulate
15                 pick stimulus l' ∈ stimuli
16                 t, l ⟵ a.tryStim(l')
17             case observe
18                 t, l ⟵ a.getObs()
19         P' ⟵ d.next(P, l)
20         if P' = ⊥ and t = U then
21             a.stop()
22             return fail
23         else
24             P ⟵ P'
25             prevLabel ⟵ l
26     a.stop()
27     return pass
```

- When l is an unexpected observation (i.e. when $P' = \bot$ and $t = \text{U}$) the test run is aborted with a **fail** verdict.

When testing continues, P is updated (at line 24) to $\mathsf{PS}(\, S \,\textbf{after}\, l\,)$, which directly corresponds to what happens in Algorithm 2.2.18. When testing stops, Algorithm 3.1 stops the IUT and returns the verdict. The actual implementation of the algorithm in our testing tool also shows the expected responses to the user.

Note that the algorithm doesn't access the suspension automaton states of the pseudo-states; all that the algorithm does with the pseudo-states, is pass them back to the DerivationEngine as argument to a DerivationEngine interface function.

3.4.4 Examples

We illustrate Algorithm 3.1 by showing two runs with the Quirky Coffee Machine specification of Figure 3.4, one on the refund-only implementation of Figure 3.5, and one on the kick-insensitive implementation of Figure 3.6. Recall that we assume implicit *input-completion* on a model that is used as implementation (see the "Implementations" paragraph of Section 3.3).

Example 1: Refund-only implementation

In Figure 3.7 we show a run with the Quirky Coffee Machine specification of Figure 3.4 on the refund-only implementation of Figure 3.5. In particular, we show the interaction between the Manager and the DerivationEngine and the Adapter. We look at a run of maximal 5 steps. We assume that the IUT makes its own choices, such that the test tool has no control over them. In Figure 3.8 on page 69 we show the transitions that are covered in the model by this run. Below, we follow the structure of Figure 3.7.

Initialisation Pseudo-state P is initialised to $\mathsf{PS}(\{0\})$, representing only the initial state of the specification (there are no internal transitions enabled in the initial state), and via the Adapter the IUT is started.

Test step 1 Set $stimuli = \mathsf{in}(\mathsf{PS}(\{0\})) = \{\mathsf{?coin}\} \neq \emptyset$. Because $stimuli \neq \emptyset$ the algorithm makes a random choice from $\{stimulate, observe\}$. Assume it chooses to observe[1]. The algorithm obtains an observation via the Adapter (line 18). The IUT is still in its initial state and has not produced output, and thus the Adapter returns $\langle \text{U}, \delta \rangle$. The label value is in $\mathsf{out}(\mathsf{PS}(\{0\})) = \{\delta\}$, and thus testing continues, with P updated to $\mathsf{PS}(\{0\}\ \mathbf{after}\ \delta) = \mathsf{PS}(\{0\})$ (i.e. it remains unchanged), and with $prevLabel = \delta$.

Test step 2 Now, the algorithm chooses to try to apply a stimulus ($stimuli \neq \emptyset$, and $prevLabel = \delta$). There is only one enabled stimulus, ?coin. The Adapter is requested to apply it, which succeeds, such that the IUT moves to state 2, and thus $l = \mathsf{?coin}$ and $t = \text{I}$ at line 19. Successor pseudo-state P' is computed: $P' = \mathsf{PS}(\{0\}\ \mathbf{after}\ \mathsf{?coin}) = \mathsf{PS}(\{1,2,3\}) \neq \bot$. Thus, testing continues with P updated to $P' = \mathsf{PS}(\{1,2,3\})$, and with $prevLabel = \mathsf{?coin}$.

Test step 3 Now, $stimuli = \mathsf{in}(\mathsf{PS}(\{1,2,3\})) = \{\mathsf{?cof}, \mathsf{?tea}, \mathsf{?kick}\} \neq \emptyset$, and $prevLabel \neq \delta$; again the algorithm makes a random choice from $\{stimulate, observe\}$. Assume this time it chooses to apply a stimulus, ?kick. However, when the Adapter receives the request to apply this stimulus, and is ready to pass it on to the IUT, the IUT decides to produce output. The Adapter thus receives an observation: !coin, which it passes on as return value of tryStim: $t = \text{U}$ and $l =$!coin. Successor pseudo-state P' is computed: $P' = \mathsf{PS}(\{1,2,3\}\ \mathbf{after}\ \mathsf{!coin}) =$

[1] This is what the implemented algorithm does in JTorX-1.8.0 on mac, with random number generator seed 547723777.

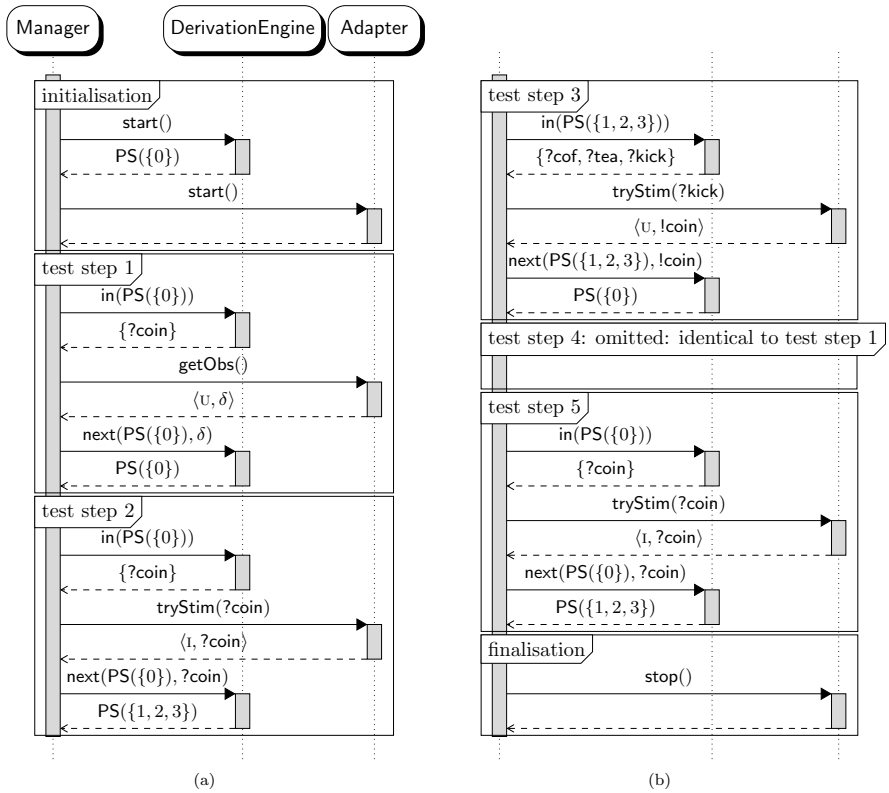


Figure 3.7: Sequence diagrams showing interaction between Manager and DerivationEngine and Adapter for the refund-only implementation.

$\mathsf{PS}(\{0\}) \neq \bot$. Thus, testing continues, with P updated to $P' = \mathsf{PS}(\{0\})$, and with $prevLabel = !\mathsf{coin}$.

Test step 4 Now $stimuli = \mathsf{in}(\mathsf{PS}(\{0\})) = \{?\mathsf{coin}\} \neq \emptyset$, and $prevLabel \neq \delta$. Assume this time the algorithm decides to observe. It requests an observation from the Adapter, which returns $\langle \mathrm{U}, \delta \rangle$, as in the first test step. Testing continues, with P updated to $\mathsf{PS}(\,\{0\}\ \mathbf{after}\ \delta\,) = \mathsf{PS}(\{0\})$ (i.e. unchanged), and with $prevLabel = \delta$.

Test step 5 Set *stimuli* has the same value as for the fourth step, but now $prevLabel = \delta$, thus the algorithm chooses to try to apply a stimulus, i.e. ?coin, the only stimulus enabled. The Adapter successfully applies it, such that the IUT moves to state 2, and $l = ?\mathsf{coin}$ and $t = \mathrm{I}$ at line 19. Successor pseudo-state P' is computed: $P' = \mathsf{PS}(\,\{0\}\ \mathbf{after}\ ?\mathsf{coin}\,) = \mathsf{PS}(\{1, 2, 3\}) \neq \bot$. Testing continues with P updated to $P' = \mathsf{PS}(\{1, 2, 3\})$, and with $prevLabel = ?\mathsf{coin}$.

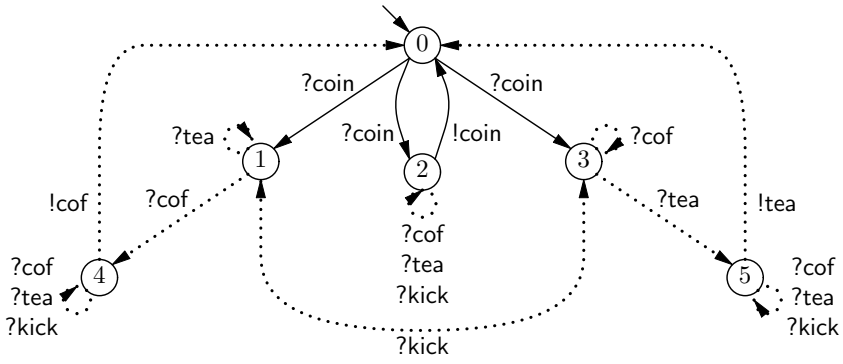


Figure 3.8: Transitions covered in quirky coffee machine model, in run of example 1, on refund-only implementation.

Finalisation Now, the expression in the condition at line 5 of the algorithm evaluates to false, the Adapter is requested to stop the IUT, and the verdict **pass** is returned.

Example 2: Kick-insensitive implementation

We now show a run with the Quirky Coffee Machine specification of Figure 3.4 on the kick-insensitive implementation of Figure 3.6. The interaction between Manager, DerivationEngine and Adapter is shown in Figure 3.9. In Figure 3.10 on page 71 we show the transitions that are covered in the model by this run.

For most of the steps, the presentation in Figure 3.9 will speak for itself. We give just a few lines of explanation for each step.

Test step 1 The algorithm has a random choice from $\{stimulate, observe\}$; we assume it chooses to stimulate with the only enabled stimulus ?coin. We assume that the IUT accepts the stimulus, randomly chooses one of the enabled transitions with label ?coin, and makes a transition to its state 3[2].

Test step 2 The algorithm again has a random choice from $\{stimulate, observe\}$; we assume it again chooses to stimulate, this time with ?cof. The IUT accepts the stimulus, and follows its only enabled ?cof transition, which is a self-loop.

Test step 3 The algorithm again has a random choice from $\{stimulate, observe\}$; we assume it chooses to obtain and check an observation. The IUT has not produced output, and does not produce output while the adapter waits, so quiescence is observed, which is correct.

[2]In JTorX we can configure two random number generator seeds: one for the test derivation functionality, and one for the simulator that is used when we use a model as IUT. With a simulator seed value of 1279935 in JTorX 1.8.0 on Mac, it makes this transition.

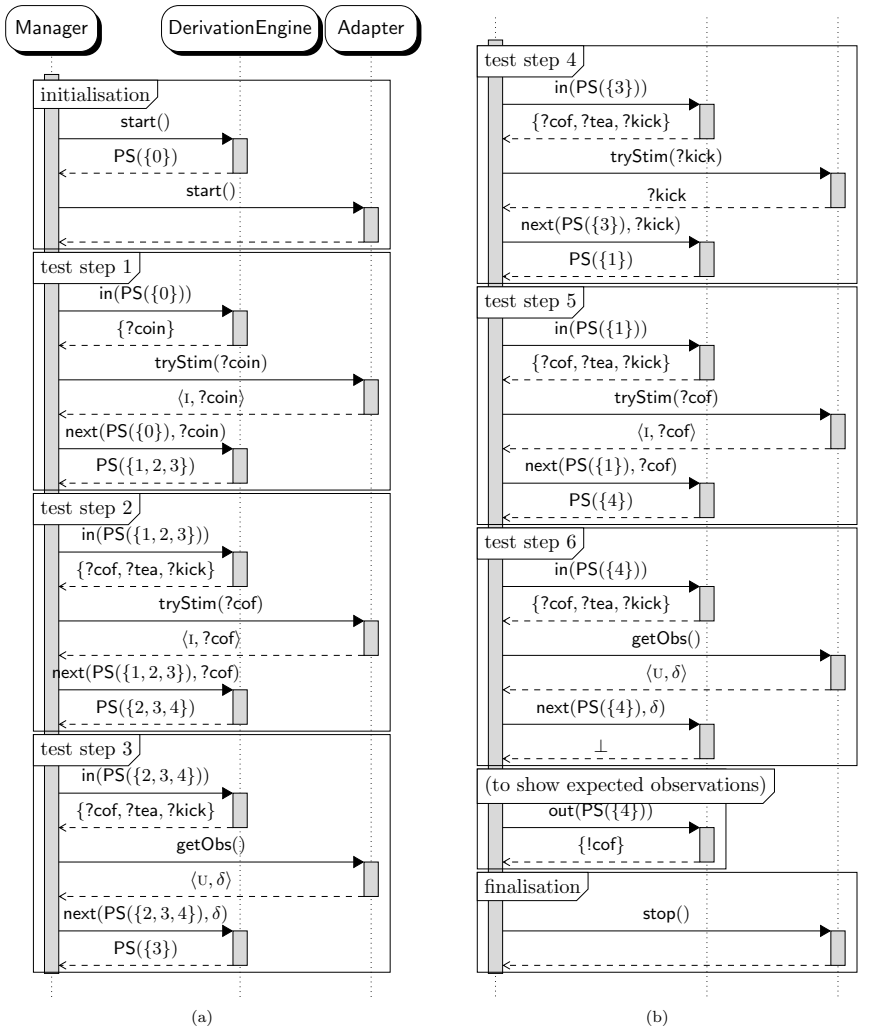


Figure 3.9: Sequence diagrams showing interaction between Manager and DerivationEngine and Adapter for the kick-insensitive implementation.

Test step 4 Having observed quiescence in the previous step, the algorithm chooses to stimulate, with, we assume, ?kick. The IUT accepts the stimulus, and follows its only enabled ?kick transition, which is a self-loop.

Test step 5 The algorithm again has a random choice from $\{stimulate, observe\}$; we assume it chooses to stimulate, this time with ?cof. The IUT accepts the stimulus, and follows its only enabled ?cof transition, which is a self-loop.

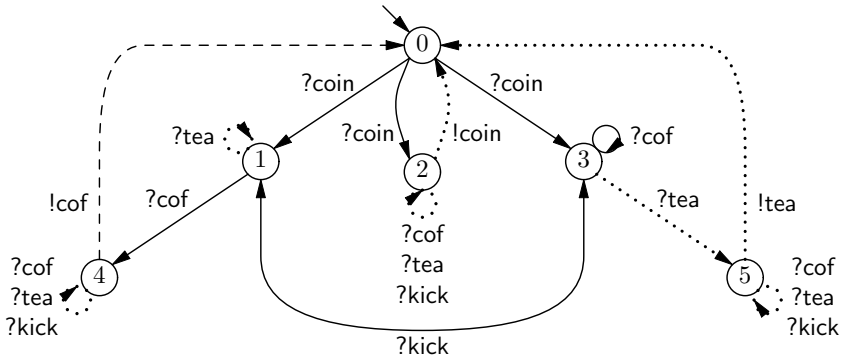


Figure 3.10: Transitions covered in quirky coffee machine model, in run of example 2, on kick-insensitive implementation. The dashed transition has the output that was expected, but not produced by the kick-insensitive implementation.

Test step 6 The algorithm again has a random choice from $\{stimulate, observe\}$; we assume it chooses to obtain and check an observation. The IUT has not produced output, and does not produce output while the adapter waits, so quiescence is observed, which is not correct. (In our testing tool implementation, the Manager requests the set of expected observations from the DerivationEngine, to show it to the user. This set is {!cof}.) Testing stops with a **fail** verdict.

3.4.5 Concluding Remarks and Observations

We conclude this section on on-line testing with the following remarks and observations.

(a) The Manager is not aware of L_I and L_U, which shows success in decoupling;

(b) Typically we see for each test step three interactions between Manager, DerivationEngine and Adapter (Figures 3.7 and 3.9 illustrate this clearly):

1. Manager obtains potential behaviour from DerivationEngine,
2. Manager interacts with Adapter for actual next step,
3. Manager updates DerivationEngine with executed next step.

There seem to be two optimisations possible.

On the one hand, it seems that we could reduce the number of interactions from three to two by combining the two interactions between Manager and DerivationEngine, i.e. by letting the DerivationEngine include the potential behaviour for the next test step in its response to the update with the last executed step. We leave this for future work.

On the other hand, the Manager can skip obtaining potential behaviour from DerivationEngine, when it knows that the Adapter has a pending

observation. However, the Adapter interface of Table 3.2 does not provide a means to pass information about pending observations from Adapter to DerivationEngine. Also this we leave for future work.

(c) For random on-line testing DerivationEngine.start, DerivationEngine.in and DerivationEngine.next suffice: we only use DerivationEngine.out to provide more information to the user, e.g. to help diagnose **fail** verdicts.

(d) By adding a simple user interface we can easily allow a user to make the choices that Algorithm 3.1 makes randomly. The user can then control the testing not unlike he would control an interactive simulator, except that during a test run we can not go one step back and explore an alternative branch (which is typically easily possible with an interactive simulator). To provide the user more insight in the testing, it is best to obtain, together with the set of possible stimuli, also the list of expected responses, and show that to the user. (For this we *do need* function DerivationEngine.out.)

3.5 Guided On-Line Testing

In the previous section we described an architecture for random on-line testing. Now we describe how this can be extended to *guidance information*, i.e. using additional information not present in the specification.

We consider two (strongly related) kinds of guidance information:

1. guidance information that *directs* the test derivation (e.g. to a particular behaviour in the specification), and
2. guidance information that *constrains* the test derivation (e.g. by imposing a particular interaction pattern).

The essential difference is that the former consists of (can be treated as) a (possibly infinite) set of finite traces, whereas the latter consists of (can be treated as) a set of *infinite* traces. In practice, however, we treat both kinds of guidance information in the same way.

In Chapter 4 we discuss a different class of guidance, namely how we can change the random test derivation such that certain actions or behaviours have a higher likely-hood to appear in the test runs than others.

Guidance based on exhibition testing We base our support for guidance on the theory for exhibition testing described in Sections 2.3 and 2.4, in particular on Algorithm 2.4.5 on page 49. In the context of this theory, the guidance information represents a singular observation objective.

Recall that we take as specification of the observation objective a set of traces from $L_\delta*$ which describe the behaviour we are interested in testing. In this setting, we get a two-dimensional verdict $\langle c, e\rangle$, with c the usual **ioco** correctness verdict (**pass** or **fail**), and e an indication of whether the implementation exhibited the observation objective (**hit** or **miss**). Thus, we may get four different verdicts:

$\langle \mathbf{pass}, \mathbf{hit}\rangle$ no error was found, and a wanted trace has been observed;
$\langle \mathbf{pass}, \mathbf{miss}\rangle$ no error was found, but no wanted trace has been observed;

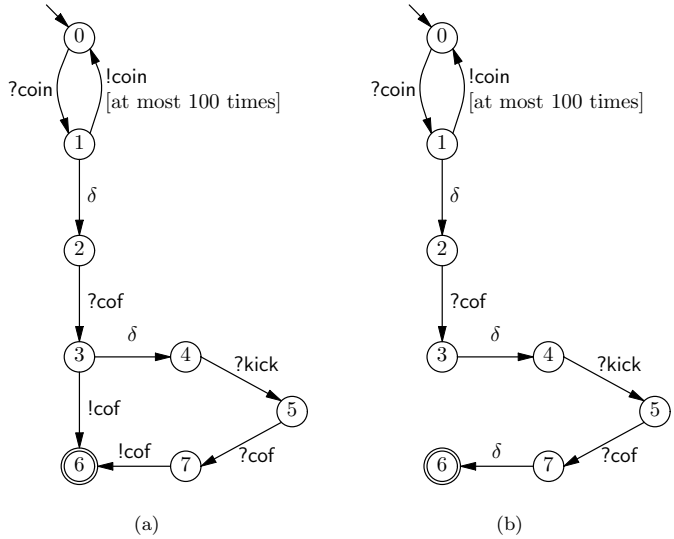


Figure 3.11: Observation objective (a) attempts to obtain coffee, either directly, or via ?kick. Observation objective (b) attempts to "hit" the error in the kick-insensitive implementation. The end states of the traces that we want to see (i.e., states that we want to reach) are marked with a double circle. In both observation objectives action !coin will be enabled only in the first 100 iterations through the loop. Thus, when the implementation returns !coin for the 101^{fst} time, action !coin will not be enabled, and the test run will end with verdict $\langle \mathbf{pass}, \mathbf{miss} \rangle$.

$\langle \mathbf{fail}, \mathbf{hit} \rangle$	an error was found, and a wanted trace has been observed;
$\langle \mathbf{fail}, \mathbf{miss} \rangle$	an error was found, but no wanted trace has been observed.

Examples We show two example observation objectives in Figure 3.11; in Section 3.5.4 we show test runs with them.

Observation objective (a) attempts to guide the test run such that coffee is obtained, either directly, or by kicking the machine. When in this way coffee is obtained, the test run will end with a verdict $\langle \mathbf{pass}, \mathbf{hit} \rangle$. Otherwise, a test run will end with either $\langle \mathbf{pass}, \mathbf{miss} \rangle$ or with $\langle \mathbf{fail}, \mathbf{miss} \rangle$. It will end with $\langle \mathbf{pass}, \mathbf{miss} \rangle$ when the goal is not reached, but no error was detected—this verdict we will get, for example, with the refund-only implementation. It will end with verdict $\langle \mathbf{fail}, \mathbf{miss} \rangle$ when an error is found, because this observation objective does not contain erroneous outputs—this we may get with the kick-insensitive implementation, because it may serve tea when coffee is requested.

Observation objective (b) attempts to guide the test run such that the error in the kick-insensitive implementation is exposed, as follows. A first attempt

to get coffee is made. When the IUT remains quiescent after this first attempt, the IUT is kicked, and a second attempt to get coffee is made. At that moment, a correct implementation should serve coffee, but the kick-insensitive implementation will remain silent. Thus, the observation objective expects quiescence at that moment. When a test run gets to that point, and observes quiescence, the run will end with a verdict $\langle \mathbf{fail}, \mathbf{hit} \rangle$, which tells us that we successfully reproduced the error scenario. Otherwise, again, a test run with observation objective (b) will end with either $\langle \mathbf{pass}, \mathbf{miss} \rangle$ or $\langle \mathbf{fail}, \mathbf{miss} \rangle$. Again, it will end with verdict $\langle \mathbf{pass}, \mathbf{miss} \rangle$ with e.g. the refund-only implementation, but also with the kick-insensitive implementation, or even the specification, when they choose the "wrong" transition when the coin is inserted. It will end with verdict $\langle \mathbf{fail}, \mathbf{miss} \rangle$ when any other error than the expected one is encountered.

About observation objectives that can lead to $\langle \mathbf{fail}, \mathbf{hit} \rangle$ Note that observation objectives that can give us $\langle \mathbf{fail}, \mathbf{hit} \rangle$ differ from the other ones, in the following way. Whereas for the other observation objectives, all traces to their goal states typically are (included in the suspension) traces of the specification s for which they are made, this is *not* the case for observation objectives that can give us $\langle \mathbf{fail}, \mathbf{hit} \rangle$. For the latter ones, the traces to their goal states are of the form $\sigma \cdot x$, with $x \in L_U^\delta$, with $\sigma \in \mathit{Straces}(s)$, and with $x \notin \mathit{out}(\, s \,\mathbf{after}\, \sigma\,)$. Thus, the last output of such trace takes us out of the specification.

3.5.1 Components

We now look at the extensions necessary to support singular observation objectives in our architecture. Figure 3.2, with the optional observation objective present, depicts the architecture.

The verdict now contains both a conformance and an exhibition aspect. We use a special DerivationEngine instance that is able to handle both a specification and a singular observation objective, and thus gives access to a richer structure than the specification-only DerivationEngine that we use for random on-line testing. Therefore, we extend the DerivationEngine interface, and we extend the Manager to use this extended DerivationEngine. As before, an Adapter provides the Manager with uniform access to the IUT.

The DerivationEngine for specification and test objective Where the specification-only DerivationEngine that we defined in Section 3.4.2 gives access to the suspension automaton of the specification (in a sense, a representation of all **ioco** test cases for the specification), this DerivationEngine gives access to a structure that is a representation of all **ioco** test cases for the specification satisfying the observation objective (like they are derived by Algorithm 2.4.5). We refer to this structure as the *exhibition automaton*. It is like a cross product of the suspension automaton of the specification and the (determinized) observation objective, with the following two special characteristics: (1) it offers the *intersection* of enabled input actions, and the *union* of enabled output actions, and (2) states in which the observation objective has been hit, or missed, are libelled with a verdict.

Although the DerivationEngine interface defined in Section 3.4 clearly offers functionality to give access to the basic elements of the exhibition automaton, extensions are necessary to give access to all information contained in it. Access to the basic elements is provided as follows: we let function in return the intersection of enabled input actions, and function out the union of enabled output actions, and functions start and next the initial state of, resp. a successor state in, the exhibition automaton. However, the Manager needs access to additional information contained in the exhibition automaton, on the one hand to let itself be guided by it; and on the other hand to be able to provide additional information to the user. The extensions give the Manager access to:

1. information to affect the choice that the Manager makes between rule 2 (try to apply stimulus) and rule 3 (obtain and check observation);
2. the verdict, if any, that is associated with an output that is returned by out (to be precise: the verdict, if any, that is associated with the state that is reached by the output);
3. the verdict, if any, that is associated with a state that is returned by start;
4. the verdict, if any, that is associated with a state that is returned by next.

We discuss each of these below.

Ad: 1: Making the Manager choose for rule 3 (observe) is trivial (and does not need extensions to the interface): the exhibition automaton contains the intersection of inputs enabled in specification and observation objective, and this is passed as return value of function in. Thus, it suffices to have no stimuli enabled in the concerning states in the observation objective: then the intersection will be empty, and the Manager will choose to observe.

Making the Manager choose for rule 2 (stimulate) is more involved: the exhibition automaton contains the union of outputs enabled in specification and observation objective. The Manager should choose to stimulate when this union is empty, or when all outputs in the union lead to **miss**. (We may be tempted to say that the Manager should choose to stimulate when the intersection of outputs enabled in specification and observation objective is empty, but that does not work for observation objectives that lead to an error, i.e. to a verdict $\langle \mathbf{fail}, \mathbf{hit} \rangle$: for such observation objective the intersection of enabled outputs is empty, and the union of enabled outputs contains at least one element that does not lead to **miss**.)

We need to extend the interface, to be able to provide the Manager with the information whether the union of enabled outputs contains one or more outputs that do not lead to **miss**.

Ad 2: When the Manager shows the enabled outputs to the user, obtained using function out, the user not only wants to see what outputs are enabled, but also, for each enabled output, the verdict (if any) that is associated with the state reached by the output. We need to extend the interface, to be able to pass such verdict information.

Ad 3: The initial state, as returned by function start, may have a verdict associated with it (e.g. when we have an observation objective that is satisfied

by the empty trace). We need to extend the interface, to be able to pass such verdicts.

Ad 4: States returned by function next may have a verdict associated with them. We need to extend the interface, to be able to pass such verdicts.

3.5.2 Interfaces

Our aim is to be able to use a single Manager (algorithm) that will work without change for both guided and unguided testing. We arrange the DerivationEngine interface functions to achieve this, as we discuss below.

Between Manager and Adapter

The interface between Manager and Adapter is unchanged w.r.t. random on-line testing.

Between Manager and DerivationEngine

Our DerivationEngine interface functions have the signature given in Table 3.5.

1.	start $:\rightarrow P \times (L_V \uplus \{\bot\})$
2.	in $: P \rightarrow \mathcal{P}(L_I)$
3.	hasOutputs $: P \rightarrow$ bool
4.	out $: P \rightarrow \mathcal{P}((L_U \cup \{\delta\}) \times (L_V \uplus \{\bot\})$
5.	next $: P \times (L_I \cup L_U \cup \{\delta\}) \rightarrow P \times (L_V \uplus \{\bot\})$
6.	defPosVerdict $:\rightarrow L_V$
7.	defNegVerdict $:\rightarrow L_V$

Table 3.5: Signature of DerivationEngine interface functions (incl. guidance), given in terms of label(set)s L_I, L_U and δ, verdicts L_V and the pseudo-state type of Table 3.6.

8.	PS $: S_\Gamma \times G_\Gamma \rightarrow P$
9.	m $: P \rightarrow S_\Gamma$
10.	g $: P \rightarrow G_\Gamma$

Table 3.6: Signature of pseudo-state type (used in guided DerivationEngine interface).

Here L_V represents the set of verdicts, i.e. $L_V = \{\langle v, e\rangle \mid v \in \{\mathbf{pass}, \mathbf{fail}\}$ and $e \in \{\mathbf{hit}, \mathbf{miss}\}\}$. A pseudo-state P now wraps a tuple of S_Γ and G_Γ, where S_Γ and G_Γ represent the set of states of Γ_s resp. Γ_g, i.e. each element of S_Γ resp. G_Γ represents a set of states of the LTS of resp. specification s and observation

objective g.

$$\begin{aligned} \mathsf{PS}(S_\Gamma, G_\Gamma).\mathsf{m}() &=_{\text{def}} S_\Gamma \\ \mathsf{PS}(S_\Gamma, G_\Gamma).\mathsf{g}() &=_{\text{def}} G_\Gamma \end{aligned}$$

We now discuss the interface functions, where we focus on the changes w.r.t. Table 3.3.

Ad 1: start Function start now returns a pseudo-state that gives access to the exhibition automaton initial state, and a verdict.

Ad 2: in Function in returns the intersection of the inputs (stimuli) enabled in specification and observation objective.

Ad 3: hasOutputs Function hasOutputs has been added to the interface, to provide the Manager with the information whether the union of enabled outputs contains one or more outputs that do not lead to **miss**.

Ad 4: out Function out returns the union of the outputs (expected observations) enabled in specification and observation objective, with, for each output returned, the verdict (if any) associated with the state reached by the output.

Ad 5: next Function next now returns a pseudo-state that gives access to the successor state in the exhibition automaton, and the verdict, if any, associated with it.

Ad 6: defPosVerdict Function defPosVerdict has been added to the interface, to allow the manager to give the right verdict, $\langle \mathbf{pass}, \mathbf{miss} \rangle$, when testing stops and no other verdict is available. This corresponds to the case where we terminate the test case with **pass** in the **ioco** algorithm.

Ad 7: defNegVerdict Function defNegVerdict has been added to the interface, to allow the manager to give the right verdict, $\langle \mathbf{fail}, \mathbf{miss} \rangle$, when we check an observation and it is expected by neither the specification nor the observation objective, i.e. when it is not in the cross product. This corresponds to the case where we terminate the test case with **fail** in the **ioco** algorithm.

Ad 8: PS Function PS is a constructor for the pseudo-state type; in the guided setting it constructs a pseudo-state from an exhibition automaton state.

Ad 9: m Function m is a pseudo-state method; it returns the specification (suspension automaton) state from a pseudo-state.

Ad 10: g Function g is a pseudo-state method; it returns the guidance (determinized observation objective) state from a pseudo-state.

Interface definition Below we define the interface functions in terms of functions in, out, out_{tr}, **after** and ϵ, of resp. Definitions 2.2.11, 2.2.10, 2.4.4, 2.2.3, and Algorithm 2.4.5. (In Chapter 4 we discuss how the interface functions can be implemented). Below s and g are the (initial states of) the specification resp. the observation objective, and S and G are sets of states (i.e. a suspension automaton state, resp. a state of a determinized LTS).

$$\mathsf{start}() =_{\text{def}} \begin{cases} \langle \mathsf{PS}(\, s \,\textbf{after}\, \epsilon \,,\, g \,\textbf{after}\, \epsilon \,), \langle \textbf{pass}, \textbf{hit} \rangle \rangle & \text{if } \epsilon(\, g \,\textbf{after}\, \epsilon \,) \\ \langle \mathsf{PS}(\, s \,\textbf{after}\, \epsilon \,,\, g \,\textbf{after}\, \epsilon \,), \bot \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{in}(P) =_{\text{def}} in(P.\mathsf{m}()) \cap in(P.\mathsf{g}())$$

$$\begin{aligned} \mathsf{out}(P) =_{\text{def}} \{ \langle l, \langle \textbf{fail}, \textbf{hit} \rangle \rangle \mid\ & l \in L_U^\delta \wedge \\ & l \notin out(P.\mathsf{m}()) \wedge \\ & l \in out_{tr}(P.\mathsf{g}()) \wedge \epsilon(\, P.\mathsf{g}() \,\textbf{after}\, l \,) \} \\ \cup \{ \langle l, \langle \textbf{pass}, \textbf{hit} \rangle \rangle \mid\ & l \in L_U^\delta \wedge \\ & l \in out(P.\mathsf{m}()) \wedge \\ & l \in out_{tr}(P.\mathsf{g}()) \wedge \epsilon(\, P.\mathsf{g}() \,\textbf{after}\, l \,) \} \\ \cup \{ \langle l, \langle \textbf{pass}, \textbf{miss} \rangle \rangle \mid\ & l \in L_U^\delta \wedge \\ & l \in out(P.\mathsf{m}()) \wedge \\ & l \notin out_{tr}(P.\mathsf{g}()) \} \\ \cup \{ \langle l, \bot \rangle \mid\ & l \in L_U^\delta \wedge \\ & l \in out(P.\mathsf{m}()) \wedge \\ & l \in out_{tr}(P.\mathsf{g}()) \wedge \not\epsilon(\, P.\mathsf{g}() \,\textbf{after}\, l \,) \} \end{aligned}$$

$$\mathsf{hasOutputs}(P) =_{\text{def}} \exists\, l \in out_{tr}(P.\mathsf{g}()) : l \in out(P.\mathsf{m}()) \vee \epsilon(\, P.\mathsf{g}() \,\textbf{after}\, l \,)$$

$$\mathsf{next}(P, l) =_{\text{def}} \begin{cases} \langle \mathsf{PS}(\, P.\mathsf{m}() \,\textbf{after}\, l \,,\, P.\mathsf{g}() \,\textbf{after}\, l \,), \langle \textbf{fail}, \textbf{hit} \rangle \rangle \\ \quad \text{if } l \in L_U^\delta \wedge l \notin out(P.\mathsf{m}()) \wedge \\ \quad\quad l \in out_{tr}(P.\mathsf{g}()) \wedge \epsilon(\, P.\mathsf{g}() \,\textbf{after}\, l \,) \\ \langle \bot, \bot \rangle \\ \quad \text{if } l \in L_U^\delta \wedge l \notin out(P.\mathsf{m}()) \wedge \\ \quad\quad (l \notin out_{tr}(P.\mathsf{g}()) \vee \not\epsilon(\, P.\mathsf{g}() \,\textbf{after}\, l \,)) \\ \langle \mathsf{PS}(\, P.\mathsf{m}() \,\textbf{after}\, l \,,\, P.\mathsf{g}() \,\textbf{after}\, l \,), \langle \textbf{pass}, \textbf{hit} \rangle \rangle \\ \quad \text{if } l \in (in(P.\mathsf{m}()) \cup out(P.\mathsf{m}())) \wedge \\ \quad\quad l \in (in(P.\mathsf{g}()) \cup out_{tr}(P.\mathsf{g}())) \wedge \\ \quad\quad \epsilon(\, P.\mathsf{g}() \,\textbf{after}\, l \,) \\ \langle \mathsf{PS}(\, P.\mathsf{m}() \,\textbf{after}\, l \,,\, P.\mathsf{g}() \,\textbf{after}\, l \,), \langle \textbf{pass}, \textbf{miss} \rangle \rangle \\ \quad \text{if } l \in (in(P.\mathsf{m}()) \cup out(S)) \wedge \\ \quad\quad l \notin (in(P.\mathsf{g}()) \cup out_{tr}(P.\mathsf{g}())) \\ \langle \mathsf{PS}(\, P.\mathsf{m}() \,\textbf{after}\, l \,,\, P.\mathsf{g}() \,\textbf{after}\, l \,), \bot \rangle \\ \quad \text{otherwise} \end{cases}$$

$\mathsf{defPosVerdict}() =_{\text{def}} \langle \mathbf{pass}, \mathbf{miss} \rangle$
$\mathsf{defNegVerdict}() =_{\text{def}} \langle \mathbf{fail}, \mathbf{miss} \rangle$

3.5.3 Manager Algorithm

We now give the algorithm that the Manager runs, as Algorithm 3.2. This algorithm differs from Algorithm 3.1, for random on-line testing, in the following

Algorithm 3.2: On-Line Test Derivation and Execution Algorithm, usable for random on-line testing and guided on-line testing.

input : A DerivationEngine d that gives access to the Specification and the observation objective, and an Adapter a that gives access to the SUT
output: A verdict

```
1  begin
2      P, v ⟵ d.start()
3      a.start()
4      prevLabel ⟵ ⊥
5      while v = ⊥ and the user does not stop the testing do
6          stimuli ⟵ d.in(P)
7          if stimuli = ∅ then
8              action ⟵ observe
9          else if prevLabel = δ or ¬d.hasOutputs(P) then
10             action ⟵ stimulate
11         else
12             action ⟵ random choice from {stimulate, observe}
13         if action = stimulate then
14             pick input l′ ∈ stimuli
15             t, l ⟵ a.tryStim(l′)
16         else obtain and check observation
17             t, l ⟵ a.getObs()
18         P′, v′ ⟵ d.next(P, l)
19         if t = U and P′ = ⊥ then
20             v ⟵ d.defNegVerdict()
21         else
22             P, v ⟵ P′, v′
23             prevLabel ⟵ l
24     a.stop()
25     if v ≠ ⊥ then
26         return v
27     return d.defPosVerdict()
```

aspects.

- the termination condition in the main loop has changed;
- we implicitly obtain a verdict at each test step;
- we use function hasOutputs to be able to force application of a stimulus; note that without guidance there will always be an expected observation, either a real one, or quiescence;
- when next returns $\langle \perp, \perp \rangle$ we give the verdict obtained with function defNegVerdict, i.e. $\langle \mathbf{fail}, \mathbf{miss} \rangle$ (was **fail**);
- when testing is stopped, and no other verdict is available we give the verdict obtained with function defPosVerdict, i.e. $\langle \mathbf{pass}, \mathbf{miss} \rangle$ (was **pass**).

Making the specification-only DerivationEngine compatible with Algorithm 3.2 The specification-only DerivationEngine (i.e. the DerivationEngine that gives access to only a specification), as defined in Section 3.4.2, is incompatible with Algorithm 3.2: it lacks the interface extensions that we made in Section 3.5.2 to support the use of guidance. For completeness sake, we list the changes that make it compatible with Algorithm 3.2:

- Add a function hasOutputs(P) that returns $out(P.\mathsf{m}()) \neq \emptyset$;
- Add a function defPosVerdict() that returns **pass**;
- Add a function defNegVerdict() that returns **fail**;
- Change function out to return, instead of a set of labels, a set of tuples $\langle l, \perp \rangle$, consisting of a label l and an empty verdict $\perp$;
- Change function init and next to return, in addition to the pseudo-state, an empty verdict $\perp$.

3.5.4 Examples

We illustrate Algorithm 3.2 by showing runs with the Quirky Coffee Machine specification of Figure 3.4. We discuss runs on the kick-insensitive machine (Fig. 3.6) and the correct machine (Fig. 3.4), both with the observation objective of Figure 3.11 (a), that tries to obtain coffee, and with the observation objective of Figure 3.11 (b), that tries to expose the error in the kick-insensitive machine. Recall that we assume implicit *input-completion* on a model that is used as implementation (see the "Implementations" paragraph of Section 3.3).

Example 1: run on kick-insensitive machine with obs. objective (a)

In this example we try to obtain coffee from the kick-insensitive machine (Figure 3.6). For this run, shown in Fig. 3.12, we assume that the IUT, upon insertion of the coin, goes to state 3, in which it is only able to produce tea. Thus, after requesting coffee, the tester observes quiescence, kicks the machine, requests coffee once more without effect, and thus a verdict $\langle \mathbf{fail}, \mathbf{miss} \rangle$ results.

For each step in the test run, the observation objective provides the action kind (apply stimulus, or obtain and check observation), and when a stimulus is to be applied, it provides the stimulus.

Note that in states 1 and 3 the observation objective allows two possible responses from the IUT: coffee and quiescence. This allows the observation

objective to deal (to a certain extent) with the non-determinism in the IUT.

Initialisation At the start of the test run d.start() is invoked. It returns a pseudo-state that gives access to the initial state $\mathsf{PS}(\{0\},\{0\})$ and verdict $\perp$.

Test step 1 Initially, $stimuli = \mathsf{in}(\mathsf{PS}(\{0\},\{0\})) = \{?\mathsf{coin}\} \cap \{?\mathsf{coin}\} = \{?\mathsf{coin}\} \neq \emptyset$. Therefore, the algorithm checks whether $prevLabel = \delta$, which is not true. Thus, it checks whether $d.\mathsf{hasOutputs}(\mathsf{PS}(\{0\},\{0\}))$ is false, which is the case, and thus it chooses to stimulate. The algorithm requests the Adapter to apply ?coin – the only available stimulus (line 15). The IUT accepts the stimulus. It randomly chooses one of the enabled transitions with label ?coin. We assume that it makes a transition to its state 3. The Adapter thus returns $\langle \text{I}, ?\mathsf{coin}\rangle$. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{0\},\{0\}), ?\mathsf{coin})$, which results in verdict $\perp$ (because ?coin is enabled in the specification and in the observation objective) and in new pseudo-state $\mathsf{PS}(\{0\}\ \mathbf{after}\ ?\mathsf{coin}, \{0\}\ \mathbf{after}\ ?\mathsf{coin}) = \mathsf{PS}(\{1,2,3\},\{1\})$ and thus testing continues, with $prevLabel$ set to ?coin.

Test step 2 Now, $stimuli = \mathsf{in}(\mathsf{PS}(\{1,2,3\},\{1\})) = \{?\mathsf{cof}, ?\mathsf{tea}, ?\mathsf{kick}\} \cap \emptyset = \emptyset$. The algorithm thus chooses to obtain and check an observation. It requests the Adapter to obtain an observation. The IUT is still in state 3 and has not produced output, and will not produce output even when the Adapter is willing to wait for it. Thus, the Adapter returns $\langle \text{U}, \delta\rangle$, the observation of quiescence. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{1,2,3\},\{1\}), \delta)$, which results in verdict $\perp$ (because δ is expected by both specification and observation objective) and in new pseudo-state $\mathsf{PS}(\{1,2,3\}\ \mathbf{after}\ \delta, \{1\}\ \mathbf{after}\ \delta) = \mathsf{PS}(\{1,3\},\{2\})$, and thus testing continues. Now, $prevLabel$ is set to δ.

Test step 3 Now, $stimuli = \mathsf{in}(\mathsf{PS}(\{1,3\},\{2\})) = \{?\mathsf{cof}, ?\mathsf{tea}, ?\mathsf{kick}\} \cap \{?\mathsf{cof}\} \neq \emptyset$. Thus, the algorithm checks whether $prevLabel = \delta$, which is true. This suffices to make the algorithm choose to stimulate. It requests the Adapter to apply the only available stimulus ?cof. The IUT still is in state 3, where the only enabled ?cof transition consists of a self-loop. It thus stays in state 3, and the Adapter returns $\langle \text{I}, ?\mathsf{cof}\rangle$. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{1,3\},\{2\}), ?\mathsf{cof})$, which results in verdict $\perp$ (because ?cof is enabled in both specification and observation objective) and in new pseudo-state $\mathsf{PS}(\{1,3\}\ \mathbf{after}\ ?\mathsf{cof}, \{2\}\ \mathbf{after}\ ?\mathsf{cof}) = \mathsf{PS}(\{3,4\},\{3\})$, and thus testing continues, with $prevLabel$ set to ?cof.

Test step 4 Now, $stimuli = \mathsf{in}(\mathsf{PS}(\{3,4\},\{3\})) = \{?\mathsf{cof}, ?\mathsf{tea}, ?\mathsf{kick}\} \cap \emptyset = \emptyset$. The algorithm thus chooses to obtain and check an observation. It requests the Adapter to obtain an observation. The IUT is still in state 3 and has not produced output, and will not produce output even when the Adapter is willing to wait for it. Thus, the Adapter returns $\langle \text{U}, \delta\rangle$. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{3,4\},\{3\}), \delta)$, which results in verdict $\perp$ (because δ is expected in both specification and observation objective) and in new pseudo-state $\mathsf{PS}(\{3,4\}\ \mathbf{after}\ \delta, \{3\}\ \mathbf{after}\ \delta) = \mathsf{PS}(\{3\},\{4\})$, and thus testing continues, with $prevLabel$ set to δ.

Test step 5 Now, $stimuli = \mathsf{in}(\mathsf{PS}(\{3\},\{4\})) = \{?\mathsf{cof}, ?\mathsf{tea}, ?\mathsf{kick}\} \cap \{?\mathsf{kick}\} \neq \emptyset$. Thus, the algorithm checks whether $prevLabel = \delta$, which is true. Assuming short-circuit evaluation of the expression, this suffices to make the algorithm choose to stimulate. (Without short-circuit evaluation the algorithm would also check whether $d.\mathsf{hasOutputs}(\mathsf{PS}(\{3\},\{4\}))$ is false, which is the case.) It requests the Adapter to apply the only available stimulus ?kick. The IUT is still in state 3. It accepts the stimulus and takes the only enabled ?kick transition, which is a self-loop, and thus it stays in state 3. The Adapter returns $\langle \text{I}, ?\mathsf{kick}\rangle$. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{3\},\{4\}), ?\mathsf{kick})$, which results in verdict $\perp$ (because ?kick is enabled in both specification and observation objective) and in new pseudo-state $\mathsf{PS}(\,\{3\}$ **after** ?kick , $\{4\}$ **after** ?kick $) = \mathsf{PS}(\{1\},\{5\})$, and thus testing continues, with *prevLabel* set to ?kick.

Test step 6 Now, $stimuli = \mathsf{in}(\mathsf{PS}(\{1\},\{5\})) = \{?\mathsf{cof}, ?\mathsf{tea}, ?\mathsf{kick}\} \cap \{?\mathsf{cof}\} \neq \emptyset$. Thus, the algorithm checks whether $prevLabel = \delta$, which is false. Thus, the algorithm checks whether $d.\mathsf{hasOutputs}(\mathsf{PS}(\{1\},\{5\}))$ is false, which is the case, and thus it chooses to apply a stimulus. It requests the Adapter to apply the only available stimulus ?cof. The IUT is in state 3. It accepts the stimulus and takes the only enabled ?cof transition, which is a self-loop, and thus it stays in state 3. The Adapter returns $\langle \text{I}, ?\mathsf{cof}\rangle$. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{1\},\{5\}), ?\mathsf{cof})$, which results in verdict $\perp$ (because ?cof is enabled in both specification and observation objective) and in new pseudo-state $\mathsf{PS}(\,\{1\}$ **after** ?cof , $\{5\}$ **after** ?cof $) = \mathsf{PS}(\{4\},\{7\})$, and thus testing continues, with *prevLabel* set to ?cof.

Test step 7 Now, $stimuli = \mathsf{in}(\mathsf{PS}(\{4\},\{7\})) = \{?\mathsf{cof}, ?\mathsf{tea}, ?\mathsf{kick}\} \cap \emptyset = \emptyset$. The algorithm thus chooses to obtain and check an observation. It requests the Adapter to obtain an observation. The IUT is still in state 3 and has not produced output, and will not produce output even when the Adapter is willing to wait for it. Thus, the Adapter returns $\langle \text{U}, \delta\rangle$. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{4\},\{7\}), \delta)$, which results in verdict $\perp$ and new pseudo-state $\perp$ because δ is expected by neither specification nor observation objective. Thus, the algorithm requests verdict $d.\mathsf{defNegVerdict}$, which is $\langle \mathbf{fail}, \mathbf{miss}\rangle$. Then, the test at line 5 evaluates to false, the Adapter is requested to stop the IUT and testing stops with verdict $\langle \mathbf{fail}, \mathbf{miss}\rangle$.

Note that the implementation may go to a different state when it receives the ?coin stimulus in the first test step, and this will give us a different test run.

Example 2: run on quirky coffee machine with obs. objective (a)

We now show a similar run on the correct quirky coffee machine, see Figure 3.13, and explain how it differs from the run on the kick-insensitive machine.

We assume that the IUT behaves in the same way, and thus the first part of the test run (up-to and including step 4) is the same as with the kick-insensitive IUT. From step 5 on, the (correct) IUT internally behaves in a different way than the kick-insensitive one, but this is not visible to the tester, until coffee is

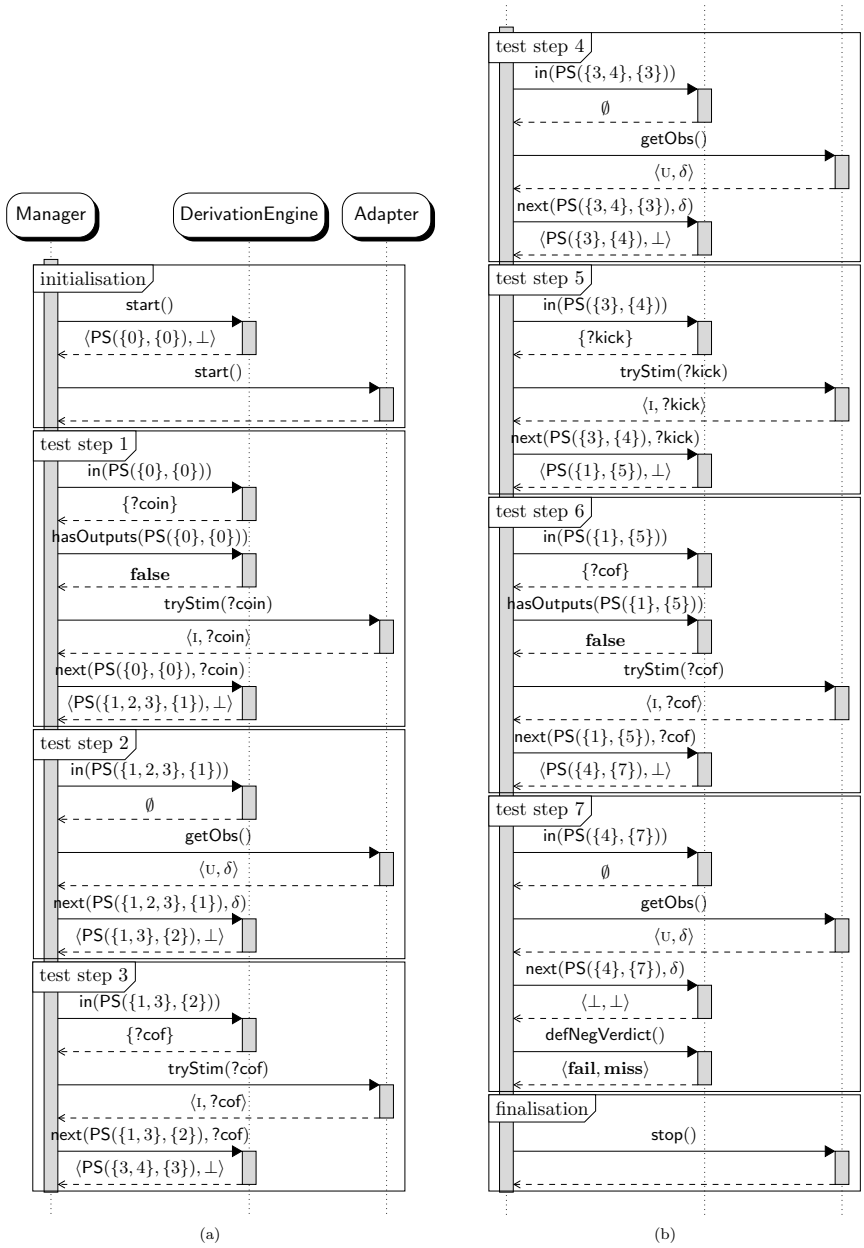


Figure 3.12: Sequence diagrams showing interaction between Manager and DerivationEngine and Adapter of guided testing the kick-insensitive machine.

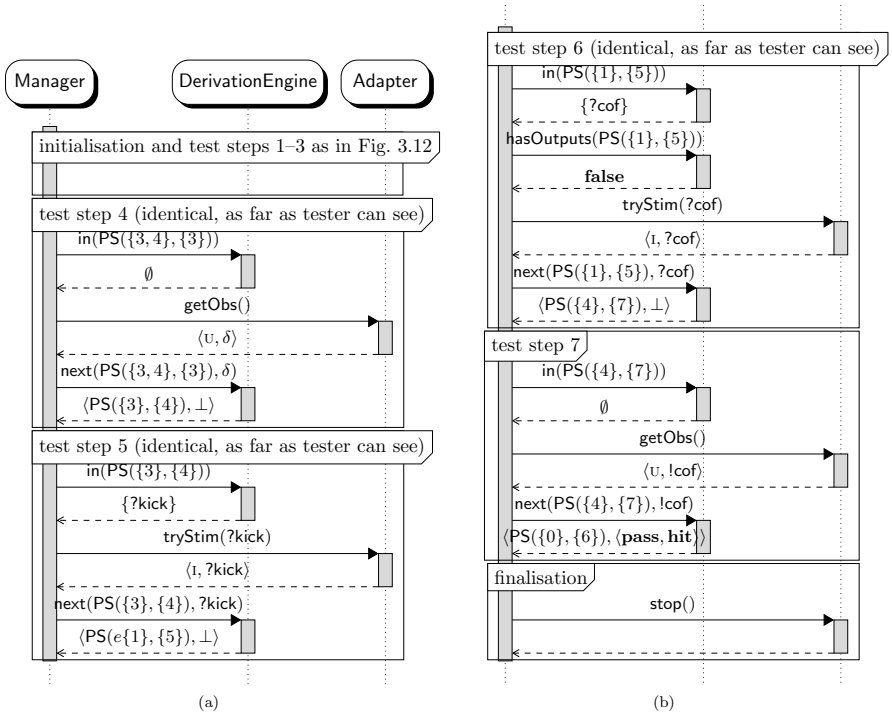


Figure 3.13: Sequence diagrams showing interaction between Manager and DerivationEngine and Adapter of guided testing the (correct) quirky coffee machine. Initialisation and first test steps have been omitted, because they are identical to the run with the kick-insensitive machine shown in Figure 3.12.

observed in step 7 (recall, with the kick-insensitive implementation, quiescence was observed in step 7). Thus, the run ends after 7 steps with verdict $\langle \mathbf{pass}, \mathbf{hit} \rangle$.

Test steps 1–4 We assume that the IUT behaves in the same way, and thus the first part of the test run is the same as with the kick-insensitive IUT.

Test step 5 In this test step the IUT is in state 3, and receives a stimulus ?kick. It accepts the stimulus and takes the only enabled ?kick transition to state 1.

Test step 6 Now, the IUT is in state 1, and it receives a stimulus ?cof . It accepts the stimulus and takes the only enabled ?cof transition to state 4. There it can produce output !cof and return to state 0. The Adapter returns $\langle \iota, ?\mathsf{cof} \rangle$. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{1\}, \{5\}), ?\mathsf{cof})$, which results in verdict $\perp$ (because ?cof is enabled in both specification and observation objective) and in new

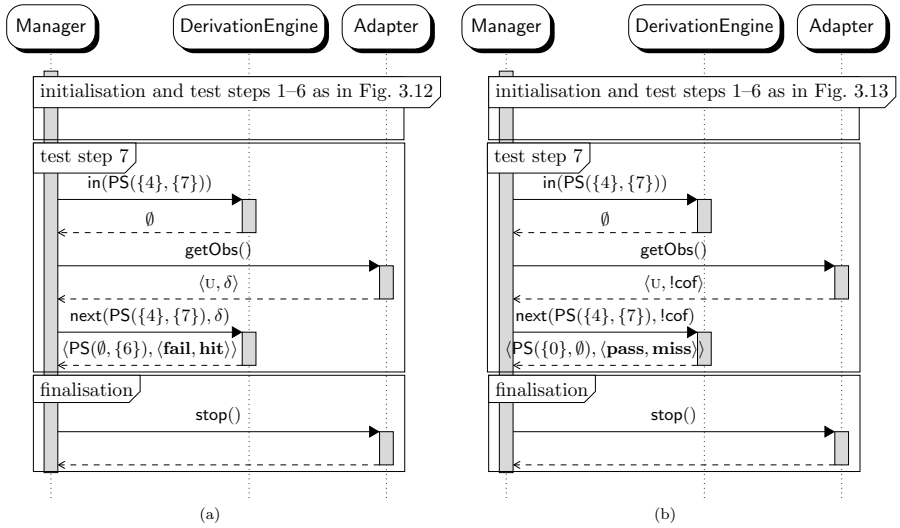


Figure 3.14: Test runs with (a) the kick-insensitive machine and (b) the (correct) quirky coffee machine, both with the observation objective of Fig. 3.11 (b) that attempts to expose the error in the kick-insensitive machine. Initialisation and first test steps have been omitted, because they are identical to the runs shown in Figure 3.12 resp. 3.13.

pseudo-state $\mathsf{PS}(\{1\}\ \textbf{after}\ ?\mathsf{cof}, \{5\}\ \textbf{after}\ ?\mathsf{cof}) = \mathsf{PS}(\{4\},\{7\})$, and thus testing continues.

Test step 7 Now, $\mathit{stimuli} = \mathsf{in}(\mathsf{PS}(\{4\},\{7\})) = \{?\mathsf{cof}, ?\mathsf{tea}, ?\mathsf{kick}\} \cap \emptyset = \emptyset$. The algorithm thus chooses to obtain and check an observation. It requests the Adapter to obtain an observation. Either the IUT is still in state 4 and has not yet produced output in which case, as mentioned above, it is willing to produce !cof and return to state 0, or it already is in state 0 and already has produced !cof. Thus, the Adapter returns $\langle \mathrm{U}, !\mathsf{cof}\rangle$. The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{4\},\{7\}), !\mathsf{cof})$, which results in verdict $\langle \textbf{pass}, \textbf{hit}\rangle$ (because !cof is expected by both specification and observation objective, and the end of the observation objective is reached – $\epsilon(\{7\}\ \textbf{after}\ !\mathsf{cof})$ holds) and new pseudo-state $\mathsf{PS}(\{4\}\ \textbf{after}\ !\mathsf{cof}, \{7\}\ \textbf{after}\ !\mathsf{cof}) = \mathsf{PS}(\{0\},\{6\})$. Thus, the test at line 19 evaluates to true, and the test at line 5 evaluates to false, after which the Adapter is requested to stop the IUT and testing stops with a $\langle \textbf{pass}, \textbf{hit}\rangle$ verdict.

Example 3: triggering error in kick-insensitive machine

Now we look at the observation objective depicted in Figure 3.11 (b). It will give us a verdict $\langle \textbf{fail}, \textbf{hit}\rangle$ when we hit the error in the kick-insensitive machine. We look at a run with the kick-insensitive machine (see Figure 3.14 (a)), and one with the (correct) quirky coffee machine (see Figure 3.14 (b)).

Test steps 1–6 We assume that the implementations behave as they did in the runs that we described as example 1 and 2. Then, with both implementations, up-to the seventh test step, the runs with this behaviour objective are as described before.

Test step 7 In the seventh test step, with both implementations, $stimuli = \emptyset$ so we decide to obtain and check an observation. Now the tester sees the difference between the two implementations.

With the kick-insensitive machine we receive an observation of δ, see Figure 3.14 (a). The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{4\},\{7\}),\delta)$, which results in verdict $\langle \mathbf{fail}, \mathbf{hit} \rangle$ (because δ is not expected by the specification but is expected by the observation objective, and the end of the observation objective is reached – $\epsilon(\,\{7\}\ \mathbf{after}\ \delta\,)$ holds) and new pseudo-state $\mathsf{PS}(\,\{4\}\ \mathbf{after}\ \delta\,,\ \{7\}\ \mathbf{after}\ \delta\,) = \mathsf{PS}(\emptyset,\{6\})$.

With the correct quirky coffee machine we receive an observation of !cof, see Figure 3.14 (b). The algorithm invokes $d.\mathsf{next}(\mathsf{PS}(\{4\},\{7\}),\mathsf{!cof})$, which results in verdict $\langle \mathbf{pass}, \mathbf{miss} \rangle$ (because !cof is expected by the specification but not by the observation objective) and new pseudo-state $\mathsf{PS}(\,\{4\}\ \mathbf{after}\ \mathsf{!cof}\,,\ \{7\}\ \mathbf{after}\ \mathsf{!cof}\,) = \mathsf{PS}(\{0\},\emptyset)$.

With both machines, testing stops after the seventh test step.

3.5.5 Concluding Remarks

Observation During testing the (state space of the) observation objective is explored on demand, as far as needed to derive the next test step (just like the specification is explored on demand). Thus, our approach will still work when we do not use as observation objective a static object that is "frozen" at the start of the test run, but an object that is dynamically extended during the test run, for example to take the effect of the current test run on the coverage of the specification into account.

Supporting User Interaction Like in the setting of random on-line testing we can allow user interaction by presenting the possible stimuli and expected observations of the next test step to the user, and let the user make the choices that otherwise are made using random choice. The user can only choose from what is enabled for the next test step. For example, if there are no inputs that are enabled in both specification and observation objective, the user is presented an empty list of stimuli, and can only choose to obtain and check an observation. To allow the user to make an informed choice, we present with each expected observation label the verdict, if available, that results from an observation with that label. (Note that the extension of out in Section 3.5.2 gives precisely this information.)

Presenting verdicts with enabled stimuli is not really necessary, because only those stimuli are presented that are enabled in both specification and observation objective. Giving a stimulus never leads to a correctness verdict in the next step, and because only those stimuli are presented that are enabled in both specification and observation objective, it also never leads to a **miss** verdict. The

only potentially interesting verdict that could be shown is a $\langle \mathbf{pass}, \mathbf{hit} \rangle$ verdict. However, such verdict would only appear when an observation objective is given that intentionally has a stimulus as last step. For such observation objectives we do not see a clear use–a typical observation objective ends with an observation. Therefore, it does not seem necessary to show verdicts with stimuli.

Optimisation for guidance that only constrains the test derivation To compute the results of DerivationEngine functions hasOutput and out the DerivationEngine has to look-ahead at the states that can be reached from the given state by doing an output action. This is clearly visible by the presence of **after** in the definition of these functions. One of the things that is checked, is whether $\epsilon(\, G \,\mathbf{after}\, l\,)$, i.e. whether we hit the end of the observation objective.

When the guidance information is meant to *constrain* the test derivation, rather than to *direct* it, the test run may **miss** the observation objective, with or without failing at the same time, but it will never **hit** the observation objective. For such observation objective, it will always be the case that $\notin(\, G \,\mathbf{after}\, l\,)$. In that case, checking whether $\epsilon(\, G \,\mathbf{after}\, l\,)$ is superfluous, and the definitions of these functions, and of function next, can be simplified. In the tool implementation we (should) have a configuration option that allows the user to indicate whether the guidance information is meant to direct the test derivation, or only to constrain it, to avoid superfluous look-ahead in the DerivationEngine.

3.6 Off-Line Test Derivation and Execution

As we have seen, the framework of Figure 2.1 gives us an initial functional decomposition of the tester into two main components: test derivation and test execution. We will now decompose these further for off-line test derivation and execution. The result of these decomposition steps is depicted in Figure 3.3, where the observation objective is optional: we will first discuss off-line test derivation without guidance, and then discuss how the presence of guidance information affects this.

We discuss off-line test derivation and execution mostly for the sake of completeness, to show that the DerivationEngine and Adapter components offer sufficient functionality to be able to do off-line testing.

As in the previous sections we first define the components of our decomposition (Section 3.6.1) and the interfaces between them (Section 3.6.2). Then, we show how these can be used to derive and execute tests. With respect to test derivation, we show four different approaches. On one hand, these approaches vary in whether tests are derived from only a specification, or also from guidance information. On the other hand, these approaches vary on how they select which test(s) to derive, and here we discuss two extremes: exhaustive, and random. The exhaustive derivation approach does not actually choose one test case over another one, but just derives all possible test cases (thus shifting the selection problem from test derivation to test execution). The random derivation approach just randomly selects a single test case to derive. Essentially, the testing then does a (number of) random walk(s) through the specification.

In Section 3.6.3 we discuss exhaustive, non-guided derivation. In Section 3.6.4 we discuss random, non-guided derivation. In Section 3.6.5 we discuss both exhaustive and random guided derivation. In Section 3.6.6 we discuss execution of individual test cases, and in Section 3.6.7 execution of test suites.

3.6.1 Components

As before we use the approach of separation of concerns to decompose both the test derivation component and the test execution component.

We decompose the test derivation component into two components: the DerivationEngine and the DerivationManager. We use the same DerivationEngine as in the previous sections (either the specification-only one, or the one that also supports guidance, depending on whether the optional observation objective is present). However, where the Manager that we have seen in previous sections directly interacts with the IUT, the DerivationManager derives test cases. Nevertheless, like the Manager, the DerivationManager is responsible for the progress of the test derivation, and it resolves the open choices in the test derivation algorithm: it decides which test case to construct.

The test execution component is very similar to our architecture for on-line testing. We decompose it in the same three components that we use there: the DerivationEngine, the ExecutionManager, and the Adapter. However, where the DerivationEngine for on-line testing gives access to a specification, the DerivationEngine here gives access to a test case, selected from the test suite—we treat a test case as a specification with a very special structure. The Adapter is the same as we have seen in previous sections. The ExecutionManager resolves most of the open choices during test execution. In this respect it is similar to the Manager for on-line testing; what is different, is that it explicitly selects test cases from the test suite. Execution of a single test case is very similar to on-line testing: the DerivationManager accesses such test via the DerivationEngine, navigates through it (as if it were a specification), and it interacts with the IUT via the Adapter, as we have seen in previous sections.

3.6.2 Interfaces

Here we discuss the interfaces between the components in our decomposition.

Between DerivationManager and ExecutionManager: the test suite The test suite forms an interface between DerivationManager and ExecutionManager. As an interface we represent it as an object, not as a function.

The formalisation of tests in Section 2.2.3 forms the basis of our test suite interface object. We make one change with respect to that formalisation: we only store the expected observations in the tests, i.e. we omit the unexpected ones. As a consequence our tests do not contain verdicts **fail** (or, in the guided case, $\langle \mathbf{fail}, \mathbf{miss} \rangle$). During test execution a **fail** (or $\langle \mathbf{fail}, \mathbf{miss} \rangle$) verdict is given when an observation is made that is not expected by the test case that is being executed. We thus lose the distinction between "known but unexpected" obser-

vations and "unknown" observations (i.e. the distinction between observation $x \in L_U^\delta, x \notin out(q)$ and $x \notin L_U^\delta, x \notin out(q)$).

We do not think this to be a great loss. This loss is compensated by the following advantages: 1) the test suite is robust against additional observations that were not in L_U when the test was derived; 2) the DerivationManager needs not be aware of L_I or L_U; and 3) the test cases are smaller when the number of elements in $out(S)$ is small compared to the number of elements in L_U.

Between DerivationManager and DerivationEngine we use the DerivationEngine interface that we gradually defined in the preceding sections.

Between ExecutionManager and Adapter we use the Adapter interface that we defined in Section 3.4.2.

Between ExecutionManager and DerivationEngine we use the DerivationEngine interface that we gradually defined in the preceding sections. In Section 3.6.6 we discuss how we use the DerivationEngine interface functions to (give) access (to) a test case.

3.6.3 Exhaustive Off-Line Derivation

In Algorithm 3.3 we sketch how we can systematically derive an exhaustive test suite by applying the rules of the **ioco** algorithm (Algorithm 3.3 uses Algo. 3.4 which in turn uses Algo. 3.5). The interfaces that we have given above suffice for this.

Recall that with the **ioco** algorithm, we start a new test with a single *placeholder*: an "open", unexpanded node (state), that is the root node of the test case that is constructed. Each of the three rules of the **ioco** algorithm turns such placeholder into either a"closed" end node (by attaching a verdict to it, i.e. **pass** in the unguided case), or an intermediate node with transitions to new placeholders (by adding outgoing transitions to it, each labelled with an input action, an output action, or δ, and each leading to a new placeholder). When the test case is ready, all placeholders have been turned into intermediate nodes, or into end nodes with a verdict.

In the case of exhaustive off-line derivation, we derive all tests of the test suite together. We try to derive them such, that each trace in the test suite has approximately the same length (i.e. same number of test steps). In each iteration of the algorithm we extend the depth of the test suite by one level (one test step). Thus, in essence the systematic exhaustive approach constructs tests in a breadth-first way—in this way our goal, that each trace in the test suite has approximately the same length, holds not only at the end of test derivation, but at any moment during test derivation. In this way, we can stop the derivation at any moment without validating our goal. In Algorithm 3.3 we continue until a sufficient depth has been reached. Other possible stop criteria include having reached the end of a finite specification, or having produced a given number of test cases.

Algorithm 3.3: Off-Line Exhaustive Test Derivation

```
   input  : A DerivationEngine d that gives access to the Specification
   output: A test suite T
 1 begin
 2     s_0, v ⟵ d.start()
 3     test suite T ⟵ {PH(s_0)}
 4     while test cases in T not deep enough do
 5         working set W ⟵ t ∈ T that have one or more placeholders
 6         T ⟵ T \ W
 7         foreach t ∈ W do
 8             // apply rules 2 and 3 from the ioco algorithm
 9             P ⟵ set of all placeholders p_i (0 ≤ i ≤ n) of t
10             // p_i.s yields the pseudo-state of placeholder p_i
11             Q ⟵ {p_i.s, for all p_i ∈ P}
12             foreach pseudo-state p_i.s ∈ Q do
13                 E_{p_i.s} ⟵ single-step-extension-set(d, p_i.s)    // Algo. 3.4
14             E_P ⟵ cartesian product E_{p_0.s} × E_{p_1.s} × ... × E_{p_n.s} for p_i ∈ P
15             foreach element e = ⟨e_0, ..., e_n⟩ in E_P do
16                 t' ⟵ clone of test case t
17                 foreach placeholder p_i in P do
18                     replace p_i in t' by corresponding extension e_i from e
19                 T ⟵ T ∪ {t'}
20     // apply rule 1 from the ioco algorithm
21     foreach t ∈ T do
22         replace all placeholders in t by pass
23     return T
```

In the algorithm, set T holds the test suite constructed so far. Typically, each test case in the test suite has one or more placeholder leaf nodes, each of which holds a reference to the corresponding pseudo-state, that in turn gives access to the corresponding suspension automaton state. We use $p = PH(s)$ to denote a placeholder that refers to pseudo-state s, and $p.s$ to denote the pseudo-state referenced by p, i.e. $PH(s).s =_{\text{def}} s$. Initially T contains a single test case that consists of a single placeholder that holds a reference to a pseudo-state that gives access to the start state of the suspension automaton.

During the test derivation, working set W holds the test cases that are to be extended during the current iteration (lines 5–19).

For each of the pseudo-states $p_i.s \in Q$ we construct the set $E_{p_i.s}$ of all possible single step extensions (lines 12–13, see Algorithm 3.4). (We can cache these to avoid having to recompute them.) Each of these single step extensions has the form of a sub-tree described in step 2 and 3 of the **ioco** algorithm, as given in Algorithm 2.2.18 on page 39. The single step extensions also contain placeholder leaf nodes, with which the corresponding suspension automaton

Algorithm 3.4: single-step-extension-set

input : A DerivationEngine d and a pseudo-state s
output: A set of single step test case extensions E

```
1  begin
2      E ⟵ ∅
3      // apply rule 3 from the ioco algorithm
4      e ⟵ obs-step-extension(d, s)                    // Algo. 3.5
5      E ⟵ E ∪ {e}
6      // apply rule 2 from the ioco algorithm on all inputs
7      e′ ⟵ e with any branch with label δ removed
8      foreach input a ∈ d.in(s) do
9          s′, v′ ⟵ d.next(s, a)
10         e ⟵ test with single branch a · PH(s′)
11         extend e with e′
12         E ⟵ E ∪ {e}
13     return E
```

Algorithm 3.5: obs-step-extension

input : A DerivationEngine d and a pseudo-state s
output: A single step test case extension e

```
1  begin
2      e ⟵ empty test case consisting of only root node
3      foreach tuple ⟨x, v⟩ ∈ d.out(s) do
4          s′, v′ ⟵ d.next(s, x)
5          extend e with branch x · PH(s′)
6      return e
```

states are associated. Now we systematically enumerate all elements in the cartesian product of the replacement sets of the placeholders of (a clone of) test case t, produce all test cases that have one level more, and store these extended test cases in T (lines 14–19).

When we have treated all test cases in W we continue in the same way to expand one more level. When we are done, we replace each placeholder of each test case in T into **pass** to obtain our test suite (lines 22–23).

Computation of the single step test case extensions In Algorithm 3.4 and Algorithm 3.5 we illustrate the computation of the set of single step test case extensions for a given pseudo-state. It constructs all possible test cases that apply a stimulus, and the single one that checks an observation. Note that, as we wrote in Section 3.6.2 when we discussed the test suite as interface, our test cases deviate from the test cases that we have defined in Chapter 2: we omit the unexpected observations (and thus, also the **fail** verdicts).

Alternative choice for information associated with placeholders In Algorithms 3.3, 3.4, and 3.5 we associate with each placeholder a pseudo-state, which we denoted as "$PH(s_0)$" and "$PH(s')$" where s_0 and s' were obtained from "$d.\mathsf{start}()$", resp. "$d.\mathsf{next}(s, a)$" and "$d.\mathsf{next}(s, x)$". As an alternative, we could associate with each placeholder, or at least each placeholder except the initial one, the information necessary to obtain the pseudo-state: the arguments to the $d.\mathsf{next}$ function. This we would denote as "$PH(s, a)$" resp. "$PH(s, x)$". This would allow us to only invoke $d.\mathsf{next}$ to compute the successor state when we are actually going to use it, i.e. when we extend the placeholder. That computation would then be done at line 11 of Algorithm 3.3 (where we obtain the pseudo-states from the placeholders).

This alternative approach has the advantage that it avoids executing $d.\mathsf{next}$ for all cases where we end a branch in the test case by replacing a placeholder by **pass** without looking at the state associated with it.

With this alternative approach we lose some traceability: for the placeholders we can no longer directly see where we are in the suspension automaton (but only after computing $d.\mathsf{next}$ on the stored information). (We thus also cannot directly see it when we return to a suspension automaton state that we have visited before.) Given that we develop this architecture and tool also to get insight in the theory by practically applying it, traceability is a property that we do not want to lose. Therefore, we will use our original approach.

3.6.4 Random Off-Line Derivation

In the previous section we gave an algorithm to systematically derive an exhaustive test suite for **ioco** from a specification, up to a given test case depth. We solved the problem of having to choose between the 3 rules of the **ioco** algorithm by *not* choosing, but instead, in a breadth-first way, generating very many test cases, such that each possible choice is represented in at least one of them.

We now discuss an alternative where we use an algorithm that derives only a single test case in each of its runs. Also here we can decide in advance (before derivation starts) on the maximal depth of the test case. Deciding on the depth of the test case resolves the question of when to apply rule 1 (end the test case). The choice between trying to apply a stimulus and checking an observation remains (except where resolved by the specification), as well as the choice of a particular stimulus.

Algorithm We give the algorithm as Algorithm 3.6. This algorithm tries to do the test derivation such that all test steps that are on the same level of the entire test case tree are of the same kind (either all are stimuli or all are expected observations). It assumes that at all moments during the derivation of the test case we can choose between stimulating and observing. However, the specification need not be input-enabled, and thus its labelled transition system representation may contain states in which no stimulus is enabled. When the algorithm wants to extend the test case with a stimulus in a branch where

Algorithm 3.6: Off-Line Random Test Derivation

```
input : A DerivationEngine d that gives access to the Specification
output: A test case t
1  begin
2      S_0, v ⟵ d.start()
3      test case t ⟵ PH(S_0)
4      while intended depth not reached do
5          switch randomly choose from {stimulate, observe} do
6              case stimulate
7                  foreach placeholder p in t do
8                      s ⟵ p.s
9                      stimuli ⟵ d.in(q)
10                     if stimuli ≠ ∅ then
11                         pick input a ∈ stimuli
12                         e ⟵ stim-step-extension(s, a)        // Algo. 3.7
13                         replace p in t by extension e
14             case observe
15                 foreach placeholder p in t do
16                     e ⟵ obs-step-extension(d, s)             // Algo. 3.5
17                     replace p in t by extension e
18     replace all placeholders in t by pass
19     return t
```

Algorithm 3.7: stim-step-extension

```
input : A pseudo-state s and stimulus a, and DerivationEngine d
output: A single step test case extension e
1  begin
2      e ⟵ test with single branch a · PH(d.next(s, a))
3      foreach tuple ⟨x, v⟩ ∈ d.out(s) where x ≠ {δ} do
4          S', v' ⟵ d.next(s, x)
5          extend e with branch x · PH(S')
6      return e
```

no stimulus is available, that branch is not extended in that iteration of the algorithm.

3.6.5 Guided Off-Line Derivation

Like we adapted the random on-line testing algorithm to take guidance into account, we also adapt the off-line derivation algorithms. Where for the on-line case we essentially follow a single trace from the trace set of a singular

Algorithm 3.8: Guided Off-Line Exhaustive Test Derivation

```
   input  : A DerivationEngine d that gives access to the Specification and
            observation objective
   output: A test suite T
 1 begin
 2     S_0, v ⟵ d.start()
 3     if v ≠ ⊥ then
 4         T ⟵ { test case consisting of verdict v}
 5         return T
 6     T ⟵ {PH(S_0)}
 7     while test cases in T still contain placeholders do
 8         working set W ⟵ t ∈ T that have one or more placeholders
 9         T ⟵ T \ W
10         foreach t ∈ W do
11             // apply rules 2 and 3 from the ioco algorithm
12             P ⟵ set of all placeholders p_i (0 ≤ i ≤ n) of t
13             // p_i.s yields the pseudo-state of placeholder p_i
14             Q ⟵ {p_i.s, for all p_i ∈ P}
15             foreach pseudo-state p_i.s ∈ Q do
16                 E_{p_i.s} ⟵ guided-single-step-extension-set(d, p_i.s)
                   // Algo. 3.9
17             E_P ⟵ cartesian product E_{p_0.s} × E_{p_1.s} × ... × E_{p_n.s} for p_i ∈ P
18             foreach element e = ⟨e_0, ..., e_n⟩ in E_P do
19                 t' ⟵ clone of test case t
20                 foreach placeholder p_i in P do
21                     replace p_i in t' by corresponding extension e_i from e
22                 T ⟵ T ∪ {t'}
23     // apply rule 1 from the ioco algorithm
24     foreach t ∈ T do replace all placeholders in t by d.defPosVerdict()
25
26     return T
```

observation objective, with off-line test derivation we can try to be exhaustive and derive a separate test case for each trace of the singular observation objective. Note, however: an observation objective that contains a loop has an infinite number of traces—in that case, the stop condition of the main loops in Algorithms 3.8 and 3.11.

Guided Exhaustive Off-Line Derivation We adapt the exhaustive off-line derivation algorithm to work with a DerivationEngine that gives access to both a specification and a singular observation objective.

We use the same approach as in the unguided case, i.e. we gradually extend the test cases. In this sense, Algorithms 3.8, 3.9 and 3.10 are guided versions

Algorithm 3.9: guided-single-step-extension-set

input : A DerivationEngine d and a suspension automaton state s
output: A set of single step test case extensions E

```
1  begin
2      E ⟵ ∅
3      // apply rule 3 from the ioco algorithm, if applicable
4      e ⟵ guided-obs-step-extension(d, s)                    // Algo. 3.10
5      if d.out(s) ≠ ∅ then
6          E ⟵ E ∪ {e}
7      // apply rule 2 from the ioco algorithm on all inputs
8      e' ⟵ e with any branch with label δ removed
9      foreach input a ∈ d.in(s) do
10         s', v' ⟵ d.next(s, a)
11         if v' = ⊥ then
12             e ⟵ test with single branch a · PH(s')
13         else
14             e ⟵ test with single branch a ·  node with verdict v'
15         extend e with e'
16         E ⟵ E ∪ {e}
17     return E
```

Algorithm 3.10: guided-obs-step-extension

input : A DerivationEngine d and a suspension automaton state s
output: A single step test case extension e

```
1  begin
2      e ⟵ empty test case consisting of only root node
3      foreach output x, v ∈ d.out(s) do
4          s', v' ⟵ d.next(s, x)
5          if v' = ⊥ then
6              extend e with branch x · PH(s')
7          else
8              extend e with branch x ·  node with verdict v'
9      return e
```

of Algorithms 3.3, 3.4 and 3.5. The main difference between the unguided and the guided algorithms is in two places.

The first place is at the stop condition of the main loop in the algorithm. The unguided algorithms need a decision when to stop the test derivation (when sufficient depth has been reached). In the guided algorithms we continue the algorithm until there are no test cases that contain placeholders—for the moment assuming that observation objectives are finite, such that we can reach the end. When we reach the end of observation objectives, we don't introduce

new placeholders, and thus at some point all placeholders that are introduced will be expanded. Note that, as stated above, we do need an additional stop condition when we are dealing with infinite observation objectives.

The second place is where we compute the single step test case extensions. In the unguided case we always put a placeholder 'at the end' of the single step extension. In the guided case we check whether the successor state computed by the DerivationEngine has a verdict, and if so, we put a node with this verdict 'at the end' of the single step extension. Only when the successor state has no verdict, we introduce a placeholder. See lines 11–14 in Algorithm 3.9, lines 5–8 in Algorithm 3.10, and lines 21–24 in Algorithm 3.11.

To extend Algorithm 3.8 to be usable both for guided and unguided testing, it is sufficient to extend the condition of the main loop (i.e. at line 7) such that we are also able to end test derivation when we have a DerivationEngine that gives access to a specification. (Note that in the algorithm for random on-line testing we do the same thing.)

Guided Random Off-Line Derivation For the random off-line derivation algorithm we essentially merge the guided random on-line testing algorithm of Algorithm 3.2 with the idea of random off-line testing of Algorithm 3.6. We show the result as Algorithm 3.11.

3.6.6 Execution of Derived Test Cases

On line testing is very similar to off-line test execution. Imagine we have a DerivationEngine that treats a test case as a special kind of specification. There are three differences between a specification and a test case.

1. A test case contains quiescence actions, and a specification does not.
2. A test case contains verdicts, and a specification does not.
3. A test case has a special structure, which a typical specification has not: in each state the test case either tries to apply a stimulus, or it obtains an observation, or it gives a verdict.

The first difference means that the DerivationEngine does not have to synthesise quiescence actions, but only has to recognise them in the test case, i.e. we use out_{tr} instead of out. The second difference means that the DerivationEngine must be able to extract verdicts from a test case and make them available to the Manager. We assume that verdicts are encoded in the test case in the same way as we encode quiescence: by self loops with special labels. The third difference means that the Manager no longer has to choose between stimulating, observing and giving a verdict, but that it has to follow the DerivationEngine. As we have seen in the discussion of guided testing, the Manager obtains this information from the DerivationEngine using the functions in and hasOutputs.

The DerivationEngine interface that we defined in Section 3.5 is rich enough to convey all information, that is contained in a the test case, to the Manager. Moreover, we can continue to use Algorithm 3.2 in the Manager. Thus, we only have to define the DerivationEngine interface functions, to be able to use our testing architecture for execution of off-line derived test cases.

Algorithm 3.11: Guided Off-Line Random Test Derivation

input : A DerivationEngine d that gives access to the Specification
output: A test case t

```
1  begin
2      s, v ⟵ PH(d.start())
3      if v ≠ ⊥ then
4          t ⟵ test case consisting of verdict v
5          return t
6      test case t ⟵ PH(s)
7      while test case t still contain placeholders do
8          foreach placeholder p in t do
9              S ⟵ p.s
10             stimuli ⟵ DerivationEngine.in(S)
11             if stimuli = ∅ then
12                 action ⟵ observe
13             else if DerivationEngine.out(S) = ∅ then
14                 action ⟵ stimulate
15             else
16                 action ⟵ random choice from {stimulate, observe}
17             if action = stimulate then
18                 e ⟵ empty test case consisting of only root node
19                 pick input a ∈ stimuli
20                 s', v' ⟵ d.next(s, a)
21                 if v' = ⊥ then
22                     extend e with branch a · PH(s')
23                 else
24                     extend e with branch a · node with verdict v'
25                 e' ⟵ guided-obs-step-extension(d, s)        // Algo. 3.10
26                 e'' ⟵ e' with any branch with label δ removed
27                 extend e with e''
28                 replace p in t by extension e
29             else action = observe
30                 e ⟵ guided-obs-step-extension(d, s)        // Algo. 3.10
31                 replace p in t by extension e
32     replace all placeholders in t by pass
33     return t
```

We assume that the test case is a (special kind of) LTS $\langle S, L_I \cup L_U^\delta \cup L_V, T, s_0 \rangle$ where the verdicts are encoded in the test case as self-loop transitions with special labels $\in L_V$.

Depending on how the test case is derived, the defPosVerdict and defNegVerdict should either return just a correctness verdict, or a tuple that also contains an exhibition verdict.

$$\mathsf{start}() =_{\text{def}} \begin{cases} \langle\, s \textbf{ after } \epsilon\,, v \rangle & \text{if } \exists v \in L_V, p \in (\, S \textbf{ after } \epsilon\,) : p \xrightarrow{v} \\ \langle\, s \textbf{ after } \epsilon\,, \bot \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{in}(S) =_{\text{def}} in(S)$$

$$\mathsf{hasOutputs}(S) =_{\text{def}} in(S) = \emptyset$$

$$\mathsf{next}(S, l) =_{\text{def}} \begin{cases} \langle\, S \textbf{ after } l\,, v \rangle \\ \quad \text{if } l \in in(S) \cup out_{tr}(S) \text{ and} \\ \quad\quad \exists v \in L_V, p \in (\, S \textbf{ after } l\,) : p \xrightarrow{v} \\ \langle\, S \textbf{ after } l\,, \bot \rangle \\ \quad \text{if } l \in in(S) \cup out_{tr}(S) \text{ and} \\ \quad\quad \nexists v \in L_V, p \in (\, S \textbf{ after } l\,) : p \xrightarrow{v} \\ \langle \bot, \bot \rangle \\ \quad \text{if } l \in L_U^{\delta} \wedge l \notin out_{tr}(S) \end{cases}$$

$$\mathsf{out}(S) =_{\text{def}} \{\langle l, v \rangle \mid l \in out_{tr}(S) \text{ and } \exists s', v : \mathsf{next}(S, l) = \langle s', v \rangle\}$$

$$\mathsf{defPosVerdict}() =_{\text{def}} \begin{cases} \langle \textbf{pass}, \textbf{miss} \rangle & \text{if the test was derived using guidance} \\ \textbf{pass} & \text{otherwise} \end{cases}$$

$$\mathsf{defNegVerdict}() =_{\text{def}} \begin{cases} \langle \textbf{fail}, \textbf{miss} \rangle & \text{if the test was derived using guidance} \\ \textbf{fail} & \text{otherwise} \end{cases}$$

3.6.7 Execution of Test Suites

In Algorithm 3.12 on the facing page we show how we can execute a test suite. We repeatedly select a test case t from the test suite T, and execute t as indicated in Section 3.6.6, until we are done. As given in Definition 2.2.14 the IUT only **passes** the test suite if each execution of a test case yields **pass** – otherwise it **fails** the suite.
This overall result will be interesting mostly when the main interest is in knowing whether a system under test conforms to a given model. If the testing is done to find, diagnose, and correct errors, then the individual test case execution results will be much more informative.

Resolving choices during test execution We have not indicated how to select the test case from the test suite, nor when to stop testing. Because we do not remove the selected test case from the test suite, we do allow the same test case to be run multiple times. When we merely want to know whether or not a IUT conforms we can stop testing as soon as a test run yields **fail**.

When no other information is available, we can resolve these choices as follows. We randomly select test cases in the test suite. Each test case that we selected we remove from the test suite. When the test suite is empty, we stop. In that way we run each test case exactly once. We can refine this by running

Algorithm 3.12: Off-Line Execution of a Given Test Suite

```
   input  : A test suite T and an Adapter a that gives access to the IUT
   output: A verdict for execution of T
1  begin
2      ready ⟵ false
3      suiteverdict ⟵ passes
4      while not ready do
5          select test case t from test suite T
6          verdict ⟵ execute t using a                    // Algo. 3.2
7          if verdict = fail then suiteverdict ⟵ fails
8      return suiteverdict
```

test cases repeatedly. We can run each selected test multiple times in succession, for example by randomly choosing between running it again and picking a next one. In that way we focus on each individual test case for a while, and then move on to the next one.

Alternatively, we can decide to only start rerunning test cases once all test cases have been run once.

When we run test cases multiple times, we have to ask ourselves whether we also want to rerun test cases that resulted in a **fail** result.

We have sketched various ways in which we can do test execution. Unfortunately we do not have a clear criterion to prefer one over the other. We have a certain preference for the approach of only rerunning test cases once all test cases have been run once, and only rerunning those test cases that not resulted in a fail verdict. We prefer this approach because it allows us to cover the functionality of the system quicker than when we run a test case multiple times before moving to the next one. Once a test case has triggered an error finding other errors first is in our opinion more useful than focusing in on the particular error. First we cover as much of the functionality as quickly as possible, to get a rough idea of where the errors are. Once we know that, we probe deeper "around" the errors, because where one error is, there probably are more.

3.7 Summary

We have decomposed our testing tool into a DerivationEngine that gives access to the specification, an Adapter that gives access to the IUT, and one (in case of on line testing) or two (in case of off-line testing) Manager components that resolve the choices that are left open in the test derivation algorithm. We have shown the testing algorithms of the Manager components, shown how they use the DerivationEngine and Adapter, and indicated how these algorithms implement the algorithms that we showed in Chapter 2.

We conclude this chapter with an overview of the interfaces of DerivationEngine and Adapter in Tables 3.7, 3.8, 3.9, 3.10, 3.11 and 3.12. In Chapter 4 and Chapter 5 we will look in more detail at DerivationEngine and Adapter.

Pseudo-state type

Signature:

PS :	$S_\Gamma \to P$
PS :	$S_\Gamma \times G_\Gamma \to P$
m :	$P \to S_\Gamma$
g :	$P \to G_\Gamma \uplus \{\bot\}$

Definition:

$\mathsf{PS}(S).\mathsf{m}$	$=_{\text{def}}$	S
$\mathsf{PS}(S,G).\mathsf{m}$	$=_{\text{def}}$	S
$\mathsf{PS}(S).\mathsf{g}$	$=_{\text{def}}$	$\bot$
$\mathsf{PS}(S,G).\mathsf{g}$	$=_{\text{def}}$	G

Table 3.7: Pseudo-state type. Each element of S_Γ represents a set of states of the LTS of specification s, and each element of G_Γ represents a set of states of the LTS of observation objective g. Furthermore, $S \in S_\Gamma$, and $G \in G_\Gamma$.

DerivationEngine:

Signature:

start :		$\to P \times (L_V \uplus \{\bot\})$
in :	P	$\to \mathcal{P}(L_I)$
hasOutputs :	P	$\to$ *bool*
out :	P	$\to \mathcal{P}((L_U \cup \{\delta\}) \times (L_V \uplus \{\bot\}))$
next :	$P \times (L_I \cup L_U \cup \{\delta\})$	$\to (P \uplus \{\bot\}) \times (L_V \uplus \{\bot\})$
defNegVerdict :		$\to L_V$
defPosVerdict :		$\to L_V$

Table 3.8: DerivationEngine Interface signature. Type P is the pseudo-state type, see Table 3.7).

DerivationEngine (continued):

Definition for unguided case:

start()	$=_{\text{def}}$	$\langle \mathsf{PS}(\, s \,\mathbf{after}\, \epsilon\,), \bot \rangle$	
in(P)	$=_{\text{def}}$	$in(P.\mathsf{m})$	
hasOutputs(P)	$=_{\text{def}}$	$out(P.\mathsf{m}) \neq \emptyset$	
out(P)	$=_{\text{def}}$	$\{\langle l, \bot \rangle \mid l \in out(P.\mathsf{m})\}$	
next(P, l)	$=_{\text{def}}$	$\begin{cases} \langle \bot, \bot \rangle & \text{if } P.\mathsf{m}\ \mathbf{after}\ l\ = \emptyset \\ \langle \mathsf{PS}(\, P.\mathsf{m}\ \mathbf{after}\ l\,), \bot \rangle & \text{otherwise} \end{cases}$	
defNegVerdict()	$=_{\text{def}}$	**fail**	
defPosVerdict()	$=_{\text{def}}$	**pass**	

Table 3.9: DerivationEngine Interface function definitions for specification-only DerivationEngine, that gives access to specification s. Type P is the pseudo-state type, see Table 3.7).

DerivationEngine (continued):
Definition for guided case:

$$
\begin{array}{lll}
\mathsf{start}() & =_{\text{def}} & \begin{cases} \langle \mathsf{PS}(\, s \,\mathbf{after}\, \epsilon\,,\ g \,\mathbf{after}\, \epsilon\,), \langle \mathbf{pass}, \mathbf{hit}\rangle\rangle \\ \quad \text{if } \epsilon(\, g \,\mathbf{after}\, \epsilon\,) \\ \langle \mathsf{PS}(\, s \,\mathbf{after}\, \epsilon\,,\ g \,\mathbf{after}\, \epsilon\,), \bot\rangle \\ \quad \text{otherwise} \end{cases} \\
\mathsf{in}(P) & =_{\text{def}} & in(P.\mathsf{m}) \cap in(P.\mathsf{g}) \\
\mathsf{hasOutputs}(P) & =_{\text{def}} & \exists\, l \in out_{tr}(P.\mathsf{g}) : l \in out(P.\mathsf{m}) \vee \\
 & & \qquad \epsilon(\, P.\mathsf{g} \,\mathbf{after}\, l\,) \\
\mathsf{out}(P) & =_{\text{def}} & \{\langle l, \langle \mathbf{fail}, \mathbf{hit}\rangle\rangle \mid l \in L_U^\delta \wedge \\
 & & \qquad l \notin out(P.\mathsf{m}) \wedge \\
 & & \qquad l \in out_{tr}(P.\mathsf{g}) \wedge \epsilon(\, P.\mathsf{g} \,\mathbf{after}\, l\,)\} \\
 & & \cup \{\langle l, \langle \mathbf{pass}, \mathbf{hit}\rangle\rangle \mid l \in L_U^\delta \wedge \\
 & & \qquad l \in out(P.\mathsf{m}) \wedge \\
 & & \qquad l \in out_{tr}(P.\mathsf{g}) \wedge \epsilon(\, P.\mathsf{g} \,\mathbf{after}\, l\,)\} \\
 & & \cup \{\langle l, \langle \mathbf{pass}, \mathbf{miss}\rangle\rangle \mid l \in L_U^\delta \wedge \\
 & & \qquad l \in out(P.\mathsf{m}) \wedge \\
 & & \qquad l \notin out_{tr}(P.\mathsf{g})\} \\
 & & \cup \{\langle l, \bot\rangle\rangle \mid l \in L_U^\delta \wedge \\
 & & \qquad l \in out(P.\mathsf{m}) \wedge \\
 & & \qquad l \in out_{tr}(P.\mathsf{g}) \wedge \not\epsilon(\, P.\mathsf{g} \,\mathbf{after}\, l\,)\} \\
\mathsf{next}(P, l) & =_{\text{def}} & \begin{cases} \langle \mathsf{PS}(\, P.\mathsf{m}() \,\mathbf{after}\, l\,,\ P.\mathsf{g}() \,\mathbf{after}\, l\,), \langle \mathbf{fail}, \mathbf{hit}\rangle\rangle \\ \quad \text{if } l \in L_U^\delta \wedge l \notin out(P.\mathsf{m}()) \wedge \\ \quad\quad l \in out_{tr}(P.\mathsf{g}()) \wedge \epsilon(\, P.\mathsf{g}() \,\mathbf{after}\, l\,) \\ \langle \bot, \bot\rangle \\ \quad \text{if } l \in L_U^\delta \wedge l \notin out(P.\mathsf{m}()) \wedge \\ \quad\quad (l \notin out_{tr}(P.\mathsf{g}()) \vee \not\epsilon(\, P.\mathsf{g}() \,\mathbf{after}\, l\,)) \\ \langle \mathsf{PS}(\, P.\mathsf{m}() \,\mathbf{after}\, l\,,\ P.\mathsf{g}() \,\mathbf{after}\, l\,), \langle \mathbf{pass}, \mathbf{hit}\rangle\rangle \\ \quad \text{if } l \in (in(P.\mathsf{m}()) \cup out(P.\mathsf{m}())) \wedge \\ \quad\quad l \in (in(P.\mathsf{g}()) \cup out_{tr}(P.\mathsf{g}())) \wedge \\ \quad\quad \epsilon(\, P.\mathsf{g}() \,\mathbf{after}\, l\,) \\ \langle \mathsf{PS}(\, P.\mathsf{m}() \,\mathbf{after}\, l\,,\ P.\mathsf{g}() \,\mathbf{after}\, l\,), \langle \mathbf{pass}, \mathbf{miss}\rangle\rangle \\ \quad \text{if } l \in (in(P.\mathsf{m}()) \cup out(S)) \wedge \\ \quad\quad l \notin (in(P.\mathsf{g}()) \cup out_{tr}(P.\mathsf{g}())) \\ \langle \mathsf{PS}(\, P.\mathsf{m}() \,\mathbf{after}\, l\,,\ P.\mathsf{g}() \,\mathbf{after}\, l\,), \bot\rangle \\ \quad \text{otherwise} \end{cases} \\
\mathsf{defNegVerdict}() & =_{\text{def}} & \langle \mathbf{fail}, \mathbf{miss}\rangle \\
\mathsf{defPosVerdict}() & =_{\text{def}} & \langle \mathbf{pass}, \mathbf{miss}\rangle
\end{array}
$$

Table 3.10: DerivationEngine Interface function definitions for DerivationEngine that gives access to specification s and observation objective g.

DerivationEngine (continued):
Definition for online execution of off-line derived tests:

$$\mathsf{start}() =_{\text{def}} \begin{cases} \langle \mathsf{PS}(\, s \textbf{ after } \epsilon\,), v\rangle \\ \quad \text{if } \exists v \in L_V, p \in (\, S \textbf{ after } \epsilon\,) : p \xrightarrow{v} \\ \langle \mathsf{PS}(\, s \textbf{ after } \epsilon\,), \bot\rangle \\ \quad \text{otherwise} \end{cases}$$

$$\mathsf{in}(P) =_{\text{def}} in(P.\mathsf{m})$$

$$\mathsf{hasOutputs}(P) =_{\text{def}} in(P.\mathsf{m}) = \emptyset$$

$$\mathsf{next}(P, l) =_{\text{def}} \begin{cases} \langle \mathsf{PS}(\, P.\mathsf{m} \textbf{ after } l\,), v\rangle \\ \quad \text{if } l \in in(P.\mathsf{m}) \cup out_{tr}(P.\mathsf{m}) \text{ and} \\ \quad\quad \exists v \in L_V, p \in (\, P.\mathsf{m} \textbf{ after } l\,) : p \xrightarrow{v} \\ \langle \mathsf{PS}(\, S \textbf{ after } l\,), \bot\rangle \\ \quad \text{if } l \in in(P.\mathsf{m}) \cup out_{tr}(P.\mathsf{m}) \text{ and} \\ \quad\quad \nexists v \in L_V, p \in (\, P.\mathsf{m} \textbf{ after } l\,) : p \xrightarrow{v} \\ \langle \bot, \bot\rangle \\ \quad \text{if } l \in L_U^{\delta} \wedge l \notin out_{tr}(P.\mathsf{m}) \end{cases}$$

$$\mathsf{out}(P) =_{\text{def}} \{\langle l, v\rangle \mid l \in out_{tr}(P.\mathsf{m}) \text{ and } \exists s', v : \mathsf{next}(P.\mathsf{m}, l) = \langle s', v\rangle\}$$

$$\mathsf{defNegVerdict}() =_{\text{def}} \begin{cases} \langle \textbf{fail}, \textbf{miss}\rangle & \text{if the test was derived using guidance} \\ \textbf{fail} & \text{otherwise} \end{cases}$$

$$\mathsf{defPosVerdict}() =_{\text{def}} \begin{cases} \langle \textbf{pass}, \textbf{miss}\rangle & \text{if the test was derived using guidance} \\ \textbf{pass} & \text{otherwise} \end{cases}$$

Table 3.11: DerivationEngine Interface function definitions for DerivationEngine that gives access to a test case s.

Adapter:
Signature:

start :		$\rightarrow$ void
tryStim :	L_I	$\rightarrow \{\text{I}, \text{U}\} \times (L_I \cup L_U)$
getObs :		$\rightarrow \{\text{U}\} \times (L_U \cup \{\delta\})$
stop :		$\rightarrow$ void

Definition:

$$\mathsf{tryStim}(a) =_{\text{def}} \begin{cases} \langle \text{U}, x \rangle \text{ with } x \in L_U & \text{if } x \text{ was produced before } a \text{ could be applied} \\ \langle \text{I}, a \rangle \text{ with } a \in L_I & \text{if } a \text{ was applied} \end{cases}$$

$$\mathsf{getObs}() =_{\text{def}} \begin{cases} \langle \text{U}, x \rangle \text{ with } x \in L_U & \text{when } x \text{ was produced by the IUT} \\ \langle \text{U}, \delta \rangle & \text{when the IUT did not produce output} \end{cases}$$

Table 3.12: Adapter Interface functions.

Chapter 4

Test Derivation Engine

In the previous chapter we gave an overview of the functionality and architecture of TorX, and we presented the interface that is offered by the DerivationEngine. In this chapter we discuss how this interface can be provided for the three cases that we have seen, i.e. for three classes of DerivationEngine instances, which differ in what they give access to:

1. only a specification (used for random testing);
2. a specification and an observation objective (used for guided testing);
3. a test case (used for test execution).

We discuss each of these classes separately (see Table 4.1). Note that they all provide the same *interface*, (i.e. they provide the same functions, with the same signature, as given in Table 3.8), but that, because each gives access to a different "input" (specification, specification + observation objective, resp. test case), they each have their own *definition (implementation)* of those functions.

used for:	Random Testing	Guided Testing	Off-Line Execution
gives access to:	specification	spec. + guide	test-case
interface def.:	Table 3.9	Table 3.10	Table 3.11
discussed in:	Sect 4.2	Sect 4.3	Sect 4.4

Table 4.1: Overview of the DerivationEngine configurations that we discuss.

Design constraints The design of each class of DerivationEngine is constrained, on the one hand, by the interface that it has to provide, as given in respectively Tables 3.9, 3.10, and 3.11, for the respective DerivationEngine classes. On the other hand, it is constrained by requirements that we discussed in Chapter 1, in particular:

1: the tool should be based on **ioco** theory;
2: the tool should work on models that have an LTS semantics;
5: the tool design should be independent from particular modelling languages;
6: the tool should support very large and infinite state space models;

9: it should be easy to accommodate theoretical progress;
10: it should be easy to incorporate new conformance relations;
13: it should be easy to create a simple model (like an automaton) for use with the tool;
14: the tool should provide insight in the theory and algorithms that it implements, e.g. by visualisation;
17: it should be possible to use the tool without being an expert in the theory that the tool implements
18: the design should allow use of modelling languages suitable for non-experts;
22: the tool should be correct.

Requirements 1, 2, 5, 6 and 14 and to a lesser extent 18, directly affect the design of the DerivationEngine, i.e. the decomposition and the inter-component interfaces; requirement 13 influences which DerivationEngine instances we create (for which modelling languages). We will come back to these requirements in the discussion of the DerivationEngine components and interfaces. We illustrate requirements 9 and 17 in our discussion of our three approaches to deal with specifications that contain τ-cycles, i.e. that are divergent. We look at requirement 10 when we discuss the specification-only DerivationEngine that is used for random testing: we first give the algorithm that allows it to be used for testing using the **ioco** implementation relation, and then we discuss the changes necessary to accommodate implementation relation **uioco**. For requirement 22 we discuss correctness of each of the DerivationEngine instances.

Remainder of this section In Section 4.1 we discuss our approaches w.r.t. dealing with (divergence due to) τ-cycles. In Section 4.2 we discuss the specification-only DerivationEngine that is used for random (unguided) testing. In Section 4.3 we extend this to the DerivationEngine that is used for guided testing, i.e. that gives access to both specification and observation objective. In Section 4.4 we discuss the DerivationEngine that is used in off-line testing, i.e. that gives access to a test case.

4.1 Dealing with τ-cycles

All DerivationEngine instances must traverse the state space of the specification, including τ-transitions, to do the work that is represented by **after** in the interface function definitions that we discussed in the previous chapter. The basic **ioco** theory that we discussed in Chapter 2 only considers specifications that are strongly converging, i.e. that do not contain τ-loops. Thus, we might choose to assume that no specification that our tool will ever have to deal with, contains a τ-cycle, such that, in our design we do not have to care about possible τ-cycles, and leave it at that. However, we think that this does not suffice, for two reasons:

1. robustness of our tool, i.e. to support careless non-expert users, and
2. theoretical progress: i.e. to incorporate extensions to the theory.

Ad 1, robustness: If we make our design with the assumption that we do not have to care about τ-cycles, we might end up with a tool that cannot handle them: it might hang when it encounters one, traversing the same τ-transition(s) over and over again. Given the fact that we want our tool to be usable by non-expert users (requirement 17), we want our tool to be able to handle "arbitrary" specifications given to it by the user—such specification may contain a τ-cycle, for example because the user has been careless, or because the user has turned observable actions into internal ones.

Thus, we want our design to be at least aware of the possibility of a τ-cycle, to avoid "hanging" when it encounters one. This gives us our first approach for dealing with τ-cycles: we just use the definitions for quiescence that we have given in the previous chapters—a state is quiescent unless it has an outgoing output- or τ-transition, and we mark quiescent states by adding a δ-self-loop to them—and merely avoid looping.

Ad 2, theoretical progress: Since the publication of the basic **ioco** theory, theoretical progress has been made. In particular, the authors of [STS13] propose an extension to the **ioco** theory to deal with divergence. With this extension, *divergent states* (states on a (fair) τ-cycle) are considered to be quiescent, because it is possible to traverse the τ-cycle forever, without providing any output, even when also output actions are enabled. (Note that the extension is defined in the framework of *Input Output Automata* (IOA), which have (1) instead of a single action τ, a set of labels denoting internal (unobservable) actions, and (2) the concept of a *task partition*, which plays a role in deciding whether a certain loop is *fair*. For the translation of our setting, we assume that the IOA that corresponds to (the LTS of) our specification has a single internal action τ, and a single task partition that contains $L_U \cup \{\tau\}$.) Below, in Section 4.1.1, we show an example specification that contains divergent states.

In [STS13] it is shown that it does not suffice to simply mark divergent states as being quiescent by adding a δ-self-loop to them: in that case it is possible that, after quiescence has been observed, output is allowed—which contradicts the intuition of quiescence (once quiescence has been observed, the system will remain quiescent until one or more stimuli are applied). (According to [STS13] this is what has been implemented in TGV [JJ05].) A solution is also provided: the δ-self-loop must not be added to the divergent states themselves, but to *copies* of them, one state copy for each divergent state. Each state copy is linked to its "original" divergent state via a δ-transition from the divergent state to the copy. Each state copy has, as outgoing transitions, in addition to the δ-self-loop, copies of the outgoing *input* transitions of the original divergent state, but not of outputs or internal actions.

Given that we want our design to able to accept theoretical improvements (requirement 9), we want our design to (also) support this approach.

Moreover, it may be instructive (remember the use in educational setting) to be able to illustrate the difference between the approach that adds δ-self-loops to the divergent states, and the one that adds them to copies of those states. Therefore, we want our design to support both these approaches.

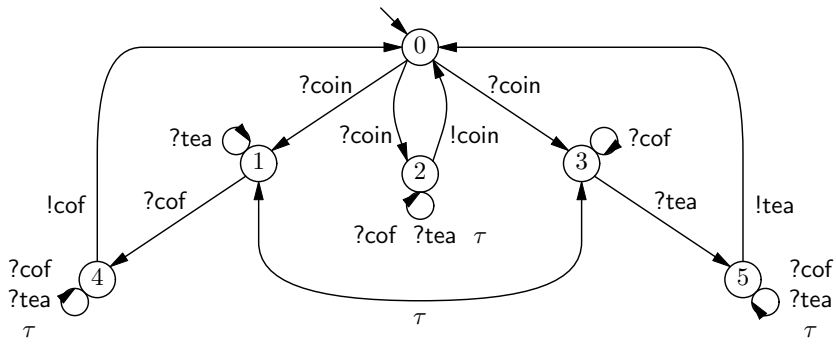


Figure 4.1: Self-kicking Coffee Machine, obtained from Quirky Coffee machine (Fig .3.4) by hiding the ?kick actions.

Supported approaches Thus, our design should support three approaches[1]:

1. just avoiding to loop on τ-cycles;
2. marking divergent states as quiescent, by adding δ-self-loops to them;
3. marking divergent states as quiescent, by adding δ-self-loops to state copies.

We discuss these approaches in more detail below, after a discussion of the variant of our quirky coffee machine that we will use to illustrate the approaches.

Note that with each of the approaches we invested only limited effort in trying to be efficient when dealing with τ-cycles, because we expect them to be rare.

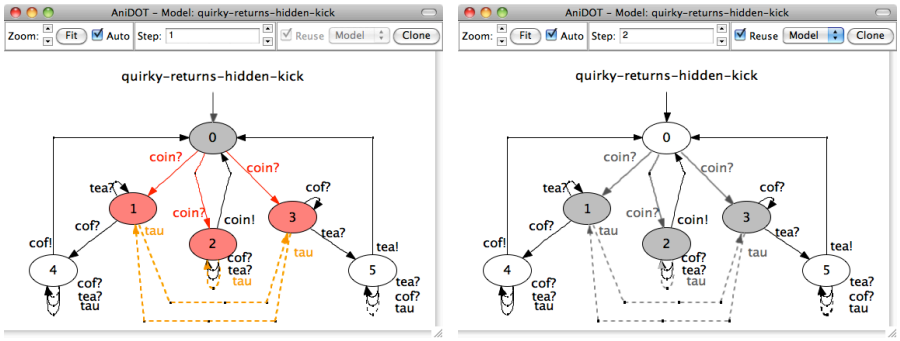
4.1.1 Example: Self-kicking Coffee Machine

As example, we need a specification that contains τ-cycles. Therefore, we took the quirky coffee machine of Figure 3.4, and replaced the ?kick actions by τ actions, which gives us the self-kicking coffee machine shown in Figure 4.1.

For the self-kicking coffee machine we have the following intuition. We assume that at the front of the machine an additional cover has been added, that prevents us from kicking the machine. We assume also that behind the machine someone is hiding (we assume a *big* machine) who keeps on kicking the machine at random moments—i.e. the machine is being kicked, but the kicking is (a) out of our control, and (b) unobservable.

In Figure 4.2 we show the visualisation of our tool (discussed in more detail in Section 4.2.5) for this specification, after execution of the first test step: application of stimulus ?coin. For each of our τ-cycle handling approaches we will look at s **after** ?coin , $out($ s **after** ?coin $)$, and where possible, at s **after** ?coin $\cdot\, \delta$.

[1] TorX implements only the first approach; JTorX implements all three, and uses the first approach by default. We are happy that JTorX implements 3 these approaches so we can experiment with them, because we are not sure (yet?) which approach we like best.



(a) (b)

Figure 4.2: (a) Self-kicking Coffee Machine after first step ?coin. The transitions that correspond with this step are highlighted (red), as are the states (s **after** ?coin) reached by these transitions. The grey markings show the previous (i.e., the initial) transition and the state(s) (s **after** ϵ) reached by it. (b) Self-kicking Coffee Machine after ?coin·δ, with the "avoid looping" approach. No states are highlighted, i.e. the last action (δ) was not in the specification: testing has stopped with a **fail** verdict. (δ-transitions are not shown).

4.1.2 Avoiding looping on τ-cycles

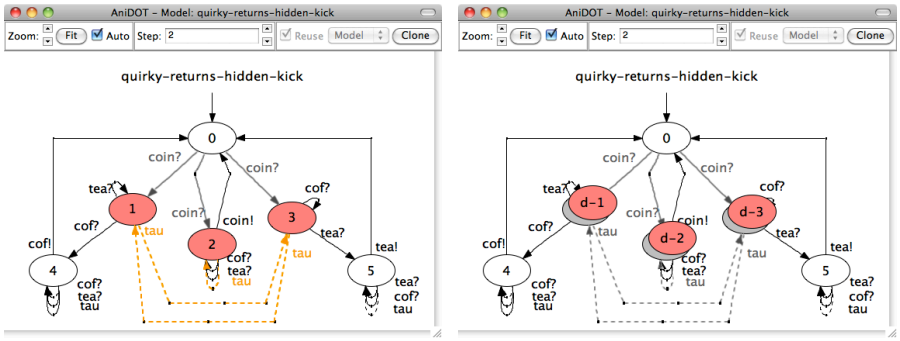
With this approach, we do not care about τ-cycles per se—we just want our testing tool to work (i.e., not hang) when a careless user gives it a specification that does contain a τ-cycle.

To avoid hanging (due to traversing the same τ-transition time and again), it suffices to recognise that an LTS state has been seen before, when we traverse the state space. We take the interpretation that a system that contains such a τ-cycle will eventually choose to do a non-τ action (i.e. we assume fairness), and thus the presence of a τ-cycle has no effect on the test that is derived: any state on the τ-cycle is not quiescent, just like any other state that has an outgoing τ-transition. We use this approach in Algorithm 4.1, the algorithm of the main DerivationEngine component, which we discuss in Section 4.2.3.

With this approach, we get the following results with self-kicking coffee machine s: s **after** ?coin $= \{1, 2, 3\}$. Moreover, $in(\{1, 2, 3\}) = \{?\mathsf{tea}, ?\mathsf{cof}\}$, and $out(\{1, 2, 3\}) = \{!\mathsf{coin}\}$. Note that states 1, 2, and 3 are not quiescent, due to their outgoing τ-transitions. Thus, if we observe quiescence after stimulus !coin, we have $\{\delta\} \not\subseteq \{!\mathsf{coin}\} = out($ s **after** ?coin), and thus we get a **fail** verdict.

4.1.3 Adding δ-self-loops to divergent states

With this approach, we reason that divergent states are quiescent. Thus, as part of the computation of quiescence, we need to detect the τ-cycles—only avoiding to loop does not suffice. We use a variant of the path-based strong component algorithm [Gab00] to find the τ-cycles, and more importantly: the (divergent!) states on them. Once the divergent states are found, with this approach we



(a) (b)

Figure 4.3: (a) Self-kicking Coffee Machine s after ?coin $\cdot\, \delta$, with the "add δ-self-loops to divergent states" approach With this approach, the state reached with ?coin $\cdot\, \delta$ is identical to the one reached with ?coin. Thus, the same states are highlighted as in Figure 4.2 (a). (δ-transitions are not shown).
(b) Self-kicking Coffee Machine after ?coin $\cdot\, \delta$, with the "add δ-self-loops to copies of divergent states" approach. With this approach, the state reached with ?coin $\cdot\, \delta$ differs from to the one reached with ?coin: the divergent state copies are selected. (incoming and outgoing transitions of the state copies are not shown.)

treat them exactly like we treat quiescent states.

In Section 4.2.7 we discuss the changes w.r.t. Algorithm 4.1, that are necessary to implement this second approach.

With self-kicking coffee machine s we get the following results. As with the previous approach, s **after** ?coin $= \{1, 2, 3\}$, and $in(\{1, 2, 3\}) = \{?\mathsf{tea}, ?\mathsf{cof}\}$.

However, now $out(\{1, 2, 3\}) = \{!\mathsf{coin}, \delta\}$. Note that with this approach states 1 and 3 are quiescent—even though they have outgoing τ-transitions—because they are on a τ-cycle. In the same way, also state 2 is quiescent, even though it has an outgoing output transition.

Now, an observation of quiescence after an (initial) stimulus ?coin is allowed: s **after** ?coin $\cdot\, \delta = \{1, 2, 3\}$. However, with the observation of quiescence we remain in the same (tester) state (i.e., we remain in the same set of LTS states), and thus,the set of expected outputs before and after observing quiescence are the same. As we saw above: $out(\{1, 2, 3\}) = \{!\mathsf{coin}, \delta\}$, and thus, with this approach an observation of !coin is allowed after an observation of quiescence.

4.1.4 Adding δ-self-loops to copies of divergent states

Like with the previous approach, with this approach we first have to identify the divergent states (in the example, when handling the states s **after** ?coin $= \{1, 2, 3\}$, the divergent states are 1, 2, and 3). Once they are found, we create copies of them (d-1, d-2, d-3 in Figure 4.3 (b)), add the δ-transition from originals to their copies, and add δ-self-loop and copies of the input transitions to the

divergent state copies (not shown in the figure). In Section 4.2.7 we discuss the changes w.r.t. the previous approach in more detail.

With self-kicking coffee machine s we get the following results. As with both other approaches, s **after** ?coin $= \{1, 2, 3\}$, and $in(\{1, 2, 3\}) = \{?\mathsf{tea}, ?\mathsf{cof}\}$. And, as with the previous approach, $out(\{1, 2, 3\}) = \{!\mathsf{coin}, \delta\}$.

However, s **after** ?coin $\cdot\, \delta = \{\text{d-1}, \text{d-2}, \text{d-3}\}$, and $out(\{\text{d-1}, \text{d-2}, \text{d-3}\}) = \{\delta\}$. Thus, directly after application of a stimulus ?coin, both an observation of δ and an observation of !coin are allowed. However, once quiescence has been observed, !coin is no longer allowed, until at least one more stimulus is applied.

4.2 DerivationEngine for Random Testing

Now we discuss the DerivationEngine for random (i.e. unguided) test derivation; Figures 3.2 and 3.3 in Chapter 3 show the position of this DerivationEngine in our architectures for on-line resp. off-line testing. In the case of random test derivation, the optional observation objective, shown in these figures, is not present: this DerivationEngine gives access to only a specification. Our design of the DerivationEngine aims to achieve the following:

1. utilise requirement 2 (models have LTS semantics) by defining the interface between the language-specific part and the rest of the DerivationEngine in terms of LTS concepts;
2. support requirement 5 (independence of modelling languages) by decomposing the DerivationEngine into language-specific and language-independent parts, such that, to support a new language, construction of a language-specific part for it suffices;
3. support requirement 5, also, by allowing the tool-user to configure which labels should be treated as input, and which as output, to support modelling languages that do not distinguish between input and output actions;
4. utilise requirement 6 (support large and infinite state models) by letting this interface provide only "on-the-fly" access to the specification—the interface provides functions to obtain the initial state, and the outgoing transitions of a given state, but not all transitions, all states, or all labels;
5. support requirement 14, by providing support for visualisation.

In the remainder of this section we first describe the decomposition (Section 4.2.1) and the inter-component interfaces (Section 4.2.2); then we describe how the main DerivationEngine component provides the DerivationEngine interface of Table 3.9 using the other DerivationEngine components, for **ioco** (Section 4.2.3), and we describe the changes necessary to accommodate **uioco** (Section 4.2.6).

4.2.1 Components

As we discussed in Section 1.2.1, we decided that for our tool design we should strive for independence of modelling languages, as expressed by requirement 5. In this way, our design should allow us to easily add support for additional modelling languages. This provides us inspiration for the decomposition of

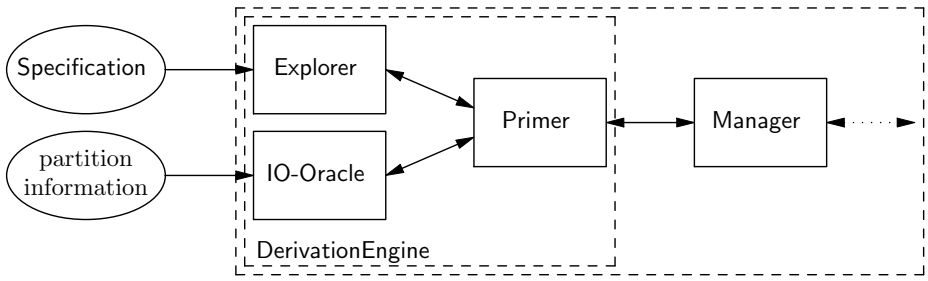


Figure 4.4: Decomposition of DerivationEngine into Explorer, Primer and IO-Oracle.

the specification-only DerivationEngine. As we show in Figure 4.4, we split the DerivationEngine in three parts: (1) an Explorer, which is specification-language specific, (2) a Primer, which is specification-language independent, and (3) an IO-Oracle, which we discuss below. Then, to provide support for an additional modelling language, we only have to create an Explorer for it.

As we discuss in Section 4.2.2, we define the interface provided by the Explorer in terms of LTS concepts. The Primer then has to realise the DerivationEngine interface, as defined in Table 3.9, using (the interfaces provided by) the Explorer and the IO-Oracle. Thus, once we have given the interface definitions and the Primer algorithm, we should be able to show that the DerivationEngine indeed realises the functionality defined in Table 3.9 (we will not actually show this, however).

The IO-Oracle: Distinguishing between input and output One issue that we have to deal with, is a mismatch between our testing theory and the language "features" of some modelling languages that we would like to use. Our testing theory assumes that we can partition the label set L of the LTS into a set of input labels L_I and a set of output labels L_U, and it assumes that there is a single representation of internal actions: τ. Some modelling languages that we would like to use do not make such distinction between inputs and outputs at the language level, or they use multiple distinct labels to represent internal actions. For such modelling languages, the information on how to partition the label set, or which labels to interpret as τ, has to come from another source: tool configuration information provided by the user. We mention two ways to deal with this.

In the approach that we have chosen, we let the Explorer return just labels $l \in L_\tau$ and let the type indication be provided by a separate IO-Oracle component in the decomposition (see Figure 4.4). The IO-Oracle is able to give the type of a given label.

An alternative approach could be to make this the responsibility of the Explorer, i.e. to let it indicate, with each label that it returns over its interface, whether the label belongs to L_I or L_U, or is the internal action τ. The Explorer interface could then pass, instead of labels, tuples $\langle l, t \rangle$ containing a label $l \in L_\tau$ together with an enumerated type $t \in \{\text{I}, \text{U}, \tau\}$ that provides such indication,

similar to the tuples returned by functions tryStrim and getObs of the Adapter interface (see Table 3.12).

We have chosen the former approach, because it allows the Explorer to be slightly simpler, and because the IO-Oracle functionality can be formalism- and thus Explorer-independent.

4.2.2 Interfaces

Inspiration for the interface between the Explorer and the Primer comes from requirements 2 and 6. By requirement 2, our design does not have to support any arbitrary modelling language: we can restrict ourself to modelling languages that have an LTS semantics. Thus, we decide that the Explorer provides an interface that gives access to an LTS. Requirement 6 states that the tool should support very large and infinite state space models. Thus, we decide that our LTS interface should not try to offer direct access to the LTS and its transition relation, but that it should allow step-by-step exploration of the LTS state space, like our DerivationEngine interface allows step-by-step exploration of the suspension automaton (in the unguided case) or exhibition automaton (in the guided case).

Between Primer and Explorer: the Explorer interface

Each Explorer must implement the following two interface functions, of which we give first the signature (in Table 4.2), and then the definitions, both in terms of an LTS $\langle S, L, T, s_0 \rangle$:

1. $\mathsf{start} :\rightarrow S$
2. $\mathsf{menu} : S \rightarrow \mathcal{P}(L_\tau \times S)$

Table 4.2: Signature of Explorer interface functions.

Ad 1: start Function start must give access to the start state of the LTS (s_0), i.e. it is defined as follows:

$$\mathsf{start}() \ =_{\text{def}} \ s_0 \tag{4.1}$$

Ad 2: menu Function menu must give access to the outgoing transitions of a given state, or more precisely, to the transition label and destination state of each outgoing transition. It is defined as follows (with $p \in S$).

$$\mathsf{menu}(p) \ =_{\text{def}} \ \{\langle l, p' \rangle \mid \exists p \xrightarrow{l} p' \in T\} \tag{4.2}$$

Between Primer and IO-Oracle: the IO-Oracle interface

The interface offered by the IO-Oracle contains only a single function, kind, which is given a label. When an IO-Oracle instance is created, it initialises itself using

the partition information; here we assume that, once initialised, the IO-Oracle represents the partition information as functions isI, isU, and isH that test whether a label represents an input, output or internal action. We don't specify how these functions are implemented—they could look up the label in a set, or look for '?' and "!" prefixes or suffixes in (the string representation of) the label, or anything else that is considered convenient. Thus, function kind has the following signature:

1. $\mathsf{kind} : L_\tau \to \{\mathrm{I}, \mathrm{U}, \tau\}$

Table 4.3: Signature of IO-Oracle interface function.

Ad 1: kind Function kind tells to what partition a given label belongs. It is defined as follows (with $l \in L_\tau$):

$$\mathsf{kind}(l) \;=_{\mathrm{def}}\; \begin{cases} \mathrm{I} & \text{if } isI(l) \\ \mathrm{U} & \text{if } isU(l) \\ \tau & \text{if } isH(l) \end{cases} \tag{4.3}$$

Provided by the Primer: the DerivationEngine interface

The Primer is the component in the DerivationEngine that realises the interface that the DerivationEngine provides to its user (i.e. to the Manager). Thus, the Primer interface coincides with the DerivationEngine interface (see Fig. 4.4). The signature of this interface was given in Table 3.8; in Section 4.2.3 below we discuss how the interface is realised.

4.2.3 Primer algorithm

We now discuss how the Primer can provide the DerivationEngine interface using the functionality offered by Explorer and IO-Oracle (we discuss the Explorer in Section 4.2.4; we do not discuss the IO-Oracle in more detail than we have done above).

To realise the interface, we refine the pseudo-state type that we introduced in the previous chapter (see Table 3.7). Recall that we can freely do so, because the user of the DerivationEngine interface is not supposed to look inside the pseudo-states. Moreover, in Chapter 3 we only mentioned constructors and methods of the pseudo-state type; we did not expose its internal structure. Below we discuss how we refine the specification-only DerivationEngine pseudo-state type, and then show the DerivationEngine interface implementation.

The specification-only DerivationEngine pseudo-state type

In Chapter 3 we mentioned constructors PS and methods m, g of the pseudo-state type. Recall that, when S is a set of LTS states, and $p = \mathsf{PS}(S)$ is a pseudo-state constructed from S, then $p.m()$ returns S, and $p.g()$ returns $\bot$.

Now we extend the pseudo-state type with four methods, of which three are used in the DerivationEngine interface implementation; the fourth (*unfold*) is used by the other three, to do the work. The four methods are:

1. $P.i()$, that returns the set of enabled inputs of P, i.e. $P.i() = in(P.m())$;
2. $P.o()$, that returns the set of tuples of an enabled output of P and a verdict ($\bot$), i.e. $P.o() = \{\langle l, \bot\rangle \mid l \in out(P.m())\}$;
3. $P.n(l)$, that returns the LTS states, reachable from P via a transition with label l, i.e. $P.n(l) = P.m()$ **after** l.
4. $P.unfold()$, that "unfolds" P (explained in detail below) and returns P (it returns P to allow the "method chaining" that we use in Table 4.4).

In Table 4.4 we show the functionality (interface) offered by the pseudo-state type, and the definitions of all its methods, except for *unfold*, which is given in Algorithm 4.1. We now first, in Definition 4.2.1, define the pseudo-state type.

Signature:	
PS :	$S_\Gamma \to P$
m :	$P \to S_\Gamma$
g :	$P \to G_\Gamma \uplus \{\bot\}$
i :	$P \to \mathcal{P}(L_I)$
o :	$P \to \mathcal{P}((L_U \cup \{\delta\}) \times (L_V \uplus \{\bot\}))$
n :	$P \times L^\delta \to S_\Gamma$
unfold :	$P \to P$
Definition:	
PS(S)	$= \langle S, \bot\rangle$ (a new pseudo-state)
P.m()	$= P.m_d \ \cup \ P.unfold().unfoldResult.m_e$
P.g()	$= \bot$
$P.i()$	$= P.unfold().unfoldResult.i$
$P.o()$	$= P.unfold().unfoldResult.o$
$P.n(l)$	$= P.unfold().unfoldResult.n_l[l]$
$P.unfold()$	= the outcome of Algo. 4.1

Table 4.4: Pseudo-state type. Each element of S_Γ represents a set of states of the LTS of specification s. Thus, $S \in S_\Gamma$ is a set of LTS states. Furthermore, P is a pseudo-state instance, and $l \in L^\delta$ is a label.

Definition 4.2.1
A pseudo-state of the specification-only Primer is a tuple $\langle m_d, unfoldResult\rangle$, with

1. m_d, a set of LTS states: those states directly reachable, like the initial state, or the destination of a set of observable transitions;
2. *unfoldResult*, a tuple $\langle i, o, n_l, m_e\rangle$ that holds the result of "unfolding" (explained below) the states in m_d. The tuple contains
   1. i, the set of enabled input labels;
   2. o, the set of tuples of an enabled output label, or δ, and a verdict ($\bot$ in the unguided case);

3. $n_l : L^\delta \to \mathcal{P}(S)$, a mapping from labels to a set that contains those states that are directly reachable via the label;
4. m_e, the set of indirectly reachable states, i.e. those states that are reachable from the directly reachable states in m_d via one or more τ transitions.

After construction of a pseudo-state instance, before it is unfolded, its *unfoldResult* will have the value $\perp$, as shown in the definition of constructor PS in Table 4.4.

□

Method *unfold*: functionality Method *unfold* provides the main functionality of the pseudo-state type. Method *unfold* returns the pseudo-state instance that it is invoked on, to allow method chaining (here used to access *unfoldResult* from the result value of *unfold*); it performs the following four tasks:

1. computing, for method $m()$, state set m_e: those states, not in the state set m_d which was given in the pseudo-state constructor, that are reachable from a state in m_d via one or more τ transitions, such that we can compute state set $Q = m_d$ **after** ϵ as $Q = m_d \cup m_e$;
2. computing, for method $i()$: $i = in(Q)$, the set of enabled input labels;
3. computing, for method $o()$: $o = \{\langle l, \perp\rangle \mid l \in out(Q)\}$, the set of tuples of an enabled output label (which includes δ if one or more of the LTS states in set Q are quiescent) and a verdict ($\perp$, in the unguided case);
4. computing, for method $n(l)$: $n_l = \{q' \mid \exists q \in Q : q \xrightarrow{l} q'\}$, the set of LTS states directly reachable by l, for each label l in the two sets i and o mentioned above.

Method *unfold* uses e.menu and IO-Oracle function kind to do its work.

Because computing e.menu may be time-consuming, we impose a constraint on how *unfold* should perform these tasks: for each Explorer LTS state q that "belongs" to a given pseudo-state Q, e.menu should be invoked only once. (But note that e.menu may be invoked multiple times on q, if q belongs to multiple different pseudo-states that are all unfolded. Of course, one could trade speed for memory by caching the results of e.menu in the Primer.)

Moreover, *unfold* must be implemented such, that, when it is invoked for a second (third, etc.) time on the same pseudo-state, it behaves as a no-op, i.e. then it only returns the pseudo-state, without doing any real work.

Method *unfold*: algorithm Method $P.unfold()$, presented in Algorithm 4.1, uses the following main data structures:

work a set of states, containing the states that have to be unfolded (initially $P.m_d$)

extra a set of states, containing those states that are found during unfolding as destination of τ-labelled transitions and that are not in $P.m_d$

inputs a set of input labels, containing the input labels encountered during unfolding

outputs a set of tuples consisting of an output label and $\perp$, containing a tuple for each output label encountered during unfolding, and for δ if a quiescent state was encountered

Algorithm 4.1: *P.unfold*() for **ioco** Primer

```
input : P, a pseudo-state
        e, an Explorer that gives access to the Specification
        o, an IO-Oracle
output: P, the pseudo-state, unfolded if it wasn't already unfolded
1  begin
2      if P.unfoldResult = ⊥ then
3          work ⟵ P.m_d
4          extra ⟵ inputs ⟵ outputs ⟵ ∅
5          destmap ⟵ {l → ∅ | l ∈ L^δ}
6          while work ≠ ∅ do
7              pick state p ∈ work
8              work ⟵ work \ {p}
9              isQuiescent ⟵ true
10             foreach ⟨l, p'⟩ ∈ e.menu(p) do
11                 kind ⟵ o.kind(l)
12                 if kind = I then
13                     inputs ⟵ inputs ∪ {l}
14                     destmap[l] ⟵ destmap[l] ∪ {p'}
15                 else if kind = U then
16                     isQuiescent ⟵ false
17                     outputs ⟵ outputs ∪ {⟨l, ⊥⟩}
18                     destmap[l] ⟵ destmap[l] ∪ {p'}
19                 else if kind = τ then
20                     isQuiescent ⟵ false
21                     if p' ∉ P.m_d ∧ p' ∉ extra then
22                         extra ⟵ extra ∪ {p'}
23                         work ⟵ work ∪ {p'}
24             if isQuiescent then
25                 outputs ⟵ outputs ∪ {⟨δ, ⊥⟩}
26                 destmap[δ] ⟵ destmap[δ] ∪ {p}
27         P.unfoldResult ⟵ ⟨inputs, outputs, destmap, extra⟩
28     return P
```

destmap : $L^\delta \to \mathcal{P}(S)$, that maps a label l to a set of states Q', where Q' contains those states that are the destination of a transition with label l (i.e. $Q' = \{q' \mid \exists q \in Q : q \xrightarrow{l} q'\}$). This is initialised such that it returns $\emptyset$ for all labels for which no specific set is assigned.

As mentioned, Algorithm 4.1 uses our "avoid-looping" approach to τ-cycles; we discuss the other two approaches in Section 4.2.7. The presence of a τ *loop* will not cause the algorithm to never finish—an *infinite sequence* of τ transitions will cause the algorithm to never finish, though.

$$\mathsf{start}() = \langle \mathsf{PS}(e.\mathsf{start}()), \bot \rangle \tag{4.4}$$

$$\mathsf{in}(P) = P.i() \tag{4.5}$$

$$\mathsf{hasOutputs}(P) = P.o() \neq \emptyset \tag{4.6}$$

$$\mathsf{out}(P) = P.o() \tag{4.7}$$

$$\mathsf{next}(P, l) = \begin{cases} \langle \bot, \bot \rangle & \text{if } P.n(l) = \emptyset \\ \langle \mathsf{PS}(P.n(l)), \bot \rangle & \text{otherwise} \end{cases} \tag{4.8}$$

$$\mathsf{defNegVerdict}() = \mathbf{fail} \tag{4.9}$$

$$\mathsf{defPosVerdict}() = \mathbf{pass} \tag{4.10}$$

Table 4.5: Implementation of the DerivationEngine interface functions, in the specification-only Primer; where e represents the Explorer that gives access to the specification.

DerivationEngine interface implementation

In Table 4.5 we show how the Primer implements the DerivationEngine interface functions, using the pseudo-state methods that we introduced above, and an Explorer e. (The IO-Oracle is not "visible" here, it is only used in $P.unfold()$.)

Correctness

Requirement 22 states: the tool should be correct. Now that we have both the definition of the DerivationEngine interface of Table 3.9, and the specification of the implementation in Table 4.5, we can discuss the correctness of the implementation.

For the implementation to be correct, the following must hold between a Primer P and the DerivationEngine interface definition D:

1. the default verdicts must be identical;
2. the pseudo-state representations returned by start must be equivalent;
3. given equivalent pseudo-state representations, in and out must return identical results;
4. given equivalent pseudo-state representations, the pseudo-state representations returned by next for any label in L^{δ}, must be equivalent again.

Note that we cannot directly compare the pseudo-states returned by the interface definition with those returned by the interface implementation. That is why we say that they must be equivalent and why we use $\sim$ in the equations below. To a certain extent we can compare pseudo-states, by comparing the LTS state sets that they represent; for this we can use method $m()$ that we defined on pseudo-states.

Our proof obligation is now to show that there exists a relation $\sim$ between concrete pseudo-states as defined here, and pseudo-states as used in the interface definition of Chapter 3, such that the following equations and condition hold:

Ad 1: The default verdicts must be identical:

$$D.\mathsf{defNegVerdict}() = P.\mathsf{defNegVerdict}() \tag{4.11}$$
$$D.\mathsf{defPosVerdict}() = P.\mathsf{defPosVerdict}() \tag{4.12}$$

Ad 2: The start states must be equivalent:

$$D.\mathsf{start}() \sim P.\mathsf{start}() \tag{4.13}$$

Ad 3: Given two equivalent pseudo-states Q_D and Q_P, i.e., such that $Q_D \sim Q_P$, the results of m, g, in, out and hasOutputs must be identical:

$$D.\mathsf{m}(Q_D) = P.\mathsf{m}(Q_P) \tag{4.14}$$
$$D.\mathsf{g}(Q_D) = P.\mathsf{g}(Q_P) \tag{4.15}$$
$$D.\mathsf{in}(Q_D) = P.\mathsf{in}(Q_P) \tag{4.16}$$
$$D.\mathsf{out}(Q_D) = P.\mathsf{out}(Q_P) \tag{4.17}$$
$$D.\mathsf{hasOutputs}(Q_D) = P.\mathsf{hasOutputs}(Q_P) \tag{4.18}$$

Ad 4: Given two equivalent pseudo-states Q_D and Q_P, obtained from D resp. P, such that $Q_D \sim Q_P$, the pseudo-states, reached via next, must be equivalent again:

$$\forall l \in L^\delta : D.\mathsf{next}(Q_D, l) \sim P.\mathsf{next}(Q_P, l) \tag{4.19}$$

Above conditions on $\sim$ are essentially the conditions of strong bisimilarity, augmented by the requirement that $\sim$-related (pseudo-)states have the same m-, g-, in-, out-, and hasOutput results.

Discussion From the definitions it is clear that Equations 4.11, 4.12, and 4.15 hold. We believe it to be possible to prove also the other equations, i.e. to prove that the suspension automata of P and D are strongly bisimilar, but we do not give that proof.

Life time of pseudo-states

We have shown creation of pseudo-states—via DerivationEngine interface functions start and next, of which we have seen applications in the algorithms in Chapter 3—but we have not discussed the life-time of these pseudo-states. (Nor have we discussed life-time of (LTS) states.) We have considered two approaches:

1. creating "fresh" instances every time, and deleting them as soon as possible;
2. creating (and reusing) instances via a factory, without deleting them.

Ad 1: creating and deleting instances For the testing algorithms that we have seen so far, this approach will work fine, especially for (random) online testing. For example, in Algorithm 3.1, we can delete the pseudo-state that represents the "current" tester state, when a new pseudo-state becomes "current" (i.e. after the assignment to P at line 24). This holds in particular, of course, when we test with models in which we do not (often) revisit states that we have visited before.

Ad 2: storing and reusing instances There are two cases where it is advantageous, or even necessary, to store and reuse pseudo-states: (a) when testing with models that have cyclic behaviour; (b) when visualising the testing.

When we have models with cyclic behaviour, i.e. where we regularly revisit states that we have visited before, the advantage of storing and reusing pseudo-states is that we do not have re-create and unfold the same pseudo-state(s) over and over again.

When we are visualising the suspension automaton, whether in its entirety, or only the part that has been (is being) traversed in a test run—both are discussed in more detail in Section 4.2.5—we want to reuse states that we have encountered (and drawn, as part of the same automaton) before: this allows us to draw loops in the suspension automaton.

Given requirement 14, our design has to support visualisation. Therefore, we have chosen to use the second approach, i.e. to use a factory to store and reuse pseudo-states. (An optimisation could be to store and reuse states when visualisation is enabled, and otherwise just create and delete states.) With the store-and-reuse approach, we have to store pseudo-states in a set or map, and then we have to decide about the identity of pseudo-states: when are two pseudo-states identical? This we discuss below.

Identity of pseudo-states

A pseudo-state "holds" a set of LTS states, and thus, it feels natural to relate the identity of a pseudo-state to the set of LTS states that it holds. (Recall that pseudo-state method $m()$ returns this set.) However, this set, and thus the identity of the pseudo-state that holds it, may undergo a transition when the pseudo-state is unfolded; we show this in an example below.

A pseudo-state P is created with an initial set of LTS states ($P.m_d$ in Definition 4.2.1, the initial state, or states directly reachable from another pseudo-state via a transition with an observable label). When P is unfolded, set $P.unfold().unfoldResult.m_e$ is computed: those states that are reachable from states in m_d via one or more τ-transitions.

In an extreme case—when the model contains one or more τ-cycles—the Primer may create pseudo-states P and Q that are (seem) distinct when they are created, but turn out to be the same state, after unfolding. Here we give an example, and discuss how we deal with this situation.

When we construct a new pseudo-state, we compare it with those that we already have, to avoid constructing state duplicates. We compare states using

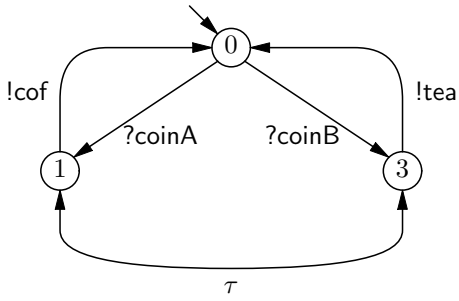


Figure 4.5: Asymmetric Machine: obtained by modifying the Self-Kicking Coffee Machine (Fig. 4.1) to illustrate the pseudo-state identify issue. Initial sets $\{1\}$ and $\{3\}$ have the same full set: $\{1, 3\}$.

a pseudo-state method *id* that is defined as follows:

$$P.id() = \begin{cases} P.m_d & \text{if } P.unfoldResult = \bot \\ P.m_d \cup P.unfoldResult.m_e & \text{otherwise} \end{cases} \tag{4.20}$$

Thus, using method *id*, the first sentence of one of the previous paragraphs can be rephrased as follows: the Primer may create pseudo-states P and Q for which $P.id() \neq Q.id()$ before unfolding, but $P.id() = Q.id()$ after unfolding. We illustrate this with the Asymmetric Machine, drawn in Figure 4.5.

Asymmetric Machine The Asymmetric Machine is a variant of the Self-Kicking Coffee Machine of Figure 4.1, modified to illustrate the pseudo-state identity issue. It only contains those transitions that we need to make our point. We call it the asymmetric, because it accepts different coins at the start, and moves to different states, depending on which coin it is given.

For the initial state of the Asymmetric Machine, accessed via an explorer e, we get a pseudo-state

$$p_0 = \mathsf{PS}(e.\mathsf{start}()) = \langle e.\mathsf{start}(), \bot \rangle = \langle \{0\}, \bot \rangle \tag{4.21}$$

Thus, $p_0.id() = \{0\}$. Unfolding this pseudo-state gives us

$$p_0 = \langle \{0\}, \langle \{?\mathsf{coinA}, ?\mathsf{coinB}\}, \{\langle \delta, \bot \rangle\}, (?\mathsf{coinA} \to \{1\}, ?\mathsf{coinB} \to \{3\}, \delta \to \{0\}), \emptyset \rangle \rangle \tag{4.22}$$

Thus, $p_0.id() = \{0\}$, also after unfolding (i.e. identical to the value before unfolding). However, things are different for the states reached from p_0 via a stimulus. For these states we get:

$$p_1 = \mathsf{PS}(p_0.n(?\mathsf{coinA})) = \langle \{1\}, \bot \rangle \tag{4.23}$$

$$p_3 = \mathsf{PS}(p_0.n(?\mathsf{coinB})) = \langle \{3\}, \bot \rangle \tag{4.24}$$

Before unfolding, $p_1.id() = \{1\}$ and $p_3.id() = \{3\}$. Unfolding gives us[2]:

$$p_1 = \langle\{1\}, \langle\emptyset, \{\langle !\mathsf{cof}, \bot\rangle, \langle !\mathsf{tea}, \bot\rangle\}, (!\mathsf{cof} \to \{0\}, !\mathsf{tea} \to \{0\}), \{3\}\rangle\rangle \quad (4.25)$$

$$p_3 = \langle\{3\}, \langle\emptyset, \{\langle !\mathsf{cof}, \bot\rangle, \langle !\mathsf{tea}, \bot\rangle\}, (!\mathsf{cof} \to \{0\}, !\mathsf{tea} \to \{0\}), \{1\}\rangle\rangle \quad (4.26)$$

Thus, after unfolding we get: $p_1.id() = \{1, 3\} = p_3.id()$.

Therefore, to identify duplicate states, we unfold a newly constructed pseudo-state before we compare it with the pseudo-states that we already have.

Moreover, to deal with the case as described above, where we have two (or more) different pseudo-states P and Q that turn out to be two different instances of effectively the same pseudo-state, we extended the pseudo-state type in our tool implementation with the following element: a reference to a "canonical" pseudo-state instance (this element is initially $\bot$, and assigned after/during unfolding). In this way, all different instances have a reference to the same canonical representation.

4.2.4 Explorer Instances

In our tool we provide a number of instances of the Explorer component, to fulfil requirement 13 (it should be easy to create a simple model (like an automaton) for use with the tool), requirement 19 (the design should allow use of modelling languages with suitable expressive power), and requirement 5 (the tool design should be independent from particular modelling languages).

An Explorer that falls in neither of these categories is the one for test run log files. Therefore, we discuss it first.

Test run log The test run log Explorer can read a test run log that is produced by the model-based testing tools TorX and JTorX (i.e., the tools that implement our design). Such log contains, for each test step, the model label that represents that test step. This Explorer treats such log as a trace of model labels.

Typical usage of this Explorer includes (i) re-running tests, (ii) use of the log to guide a subsequent test, and (iii) to extract the trace from the log and save it in Aldebaran format (discussed below), for use with other tools.

When the Explorer is used to re-run a test, it effectively treats the log as a model, to be simulated as SUT. This allows to observe e.g. the effects of changes to a model, or changes to the Primer component.

Easy creation of simple models

To support requirement 13, our tool has built-in Explorer functionality for three (modelling) languages:

1. Aldebaran (`.aut`),
2. GraphML (`.graphml`), and
3. GraphViz DOT (`.gv`).

[2]Here we assume the "avoid-looping" approach was used to deal with the τ-loop.

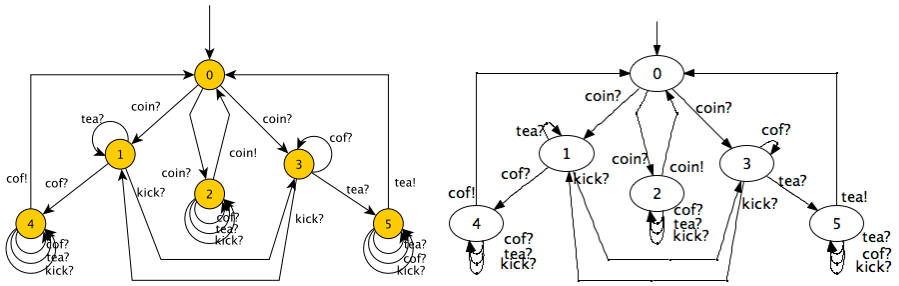


Figure 4.6: GRAPHML representation of the quirky coffee machine of Figure 3.4, created with YED (a). Our visualisation of this GRAPHML representation (b).

Aldebaran The ALDEBARAN language [ald] is a simple textual file format that allows specification of a (finite) LTS. For small models, ALDEBARAN files are typically created by hand; however, there is also quite a number of tools that are able to export an LTS in ALDEBARAN format.

The ALDEBARAN format consists of a header and a body. The header contains the identity (number) of the initial state, the number of transitions, and the number of states. The body consists of a list of transitions, with for each transition the identity (number) of the source state, the transition label, and the identity (number) of the destination state.

GraphML The GRAPHML language [Gra12] is an XML file format that allows specification of a graph.This format is (one of) the native format(s) of YED [yWo], a graph editor that makes it simple to draw a graph representation of an automaton (or LTS). The GRAPHML format not only contains nodes, edges and edge labels (that we interpret as states, transitions and transition labels) but also (a.o.) node labels and position information of nodes, edges and edge labels, and other visual information (colours, fonts, visibility, etc.). From a GRAPHML file, the GRAPHML Explorer obtains, in addition to the LTS information (nodes, edges, edge labels), also the node labels and the position information (and associates this with the LTS information). The node labels and position information are used for the visualisation (discussed later in this section), to make the visualisation of GRAPHML models resemble as much as possible the layout created in YED (see Figure 4.6).

The GRAPHML format, unfortunately, does not have the concept of an initial state. Our GRAPHML Explorer allows specification of the initial state in two ways:

1. via an initial transition (of which there must be at most one): an edge without label, of which the destination is interpreted as initial state;
2. by giving it node label "1" (with YED, the first node drawn in a new graph always has this label (this is used when no initial transition is present in the graph).

The GraphML Explorer in JTorX reuses the GraphML parser that was developed by Lars Frantzen for his iocoChecker tool (iocoChecker is discussed in Appendix A.2).

DOT The GraphViz DOT language [Graa] is a textual file format that allows specification of graphs. It is the input language for the tools in the GraphViz toolset [Grab]; typically, these tools are used to automatically create layouts of graphs, of which only the structure (nodes, edges, node labels, edge labels) is given.

Like the GraphML format, the GraphViz language does not have the concept of an initial state. Like the GraphML Explorer, the GraphViz Explorer tries to identify the initial state by recognising a (unique) initial transition.

Modelling Languages with Suitable Expressive Power

To support requirement 19, our tool allows the use of external Explorer components. (At the same time, this fulfils requirement 5.) Such external Explorer components communicate with our testing tool via the *torx-explorer* interface [torb]. This interface offers (an extension of) the functionality of the Explorer interface of Table 4.2: it also can pass information for visualisation—we return to this in Section 4.2.5.

Table 4.6 gives an (non-exhaustive) overview of the tools and toolsets for which the *torx-explorer* interface has been implemented, indicating the modelling languages that each of them supports.

tool / toolset	*language(s)*
Jararaca	Jararaca input language (`.jrrc`)
STSimulator	STS (`.sax`)
Ta2Torx	Timed Automaton
SPEX	Promela
CADP	LOTOS, BCG (`.bcg`), *others*
LTSmin	mCRL2, *others*
mCRL2	μCRL, mCRL2

Table 4.6: Overview of tools (top) and toolsets (bottom) for which the *torx-explorer* interface has been implemented, indicating the modelling languages that they provide access to.

Jararaca The Jararaca language [Jar12] allows the concise specification of (a set of) traces using a notation that was inspired by regular expressions. It was developed to have an easy notation for the specification of test purposes (observation objectives). A Jararaca file consists of 4 sections: (1) a free-format textual description; (2) a list of named action labels that are used in the test purpose (the action names are used in the regular expressions as shorthand for labels); (3) a list of named regular expressions (the name allows referring to

a preceding regular expression); (4) the "goal" regular expression that describes the (set of) traces (typically, referring to named regular expressions).

The JARARACA tool builds an automaton for the given input, and provides access to it via the *torx-explorer* interface. For visualisation, the JARARACA tool can output the automaton in DOT format (the automaton is more abstract than the corresponding LTS: a single automaton node or edge may have multiple corresponding LTS states and transitions). This automaton in DOT format is only used for display, not to access the LTS of the automaton—the LTS is accessed via the *torx-explorer* interface. However, the automaton in DOT format contains node and edge identifiers. When state and transition information is passed over the *torx-explorer* interface, also the corresponding node and edge identifiers are passed, to allow highlighting of the right visual elements (nodes and edges) in the visualisation.

Both the JARARACA tool and its input language were designed and implemented by René de Vries.

STSimulator The STSIMULATOR tool provides access to specifications in the Symbolic Transition System (STS) language. An STS is an automaton with state variables, parameters in the transition labels, and conditions and state variable updates on the transitions. The conditions can refer to state variables and label parameters; the updates assign new values to state variables. We discuss this in greater detail in Chapter 6.

For visualisation, the STSIMULATOR tool can output the STS in DOT format. Like with the JARARACA tool, node and edge identifiers that are used in the DOT format representation are passed with the state and transition information over the *torx-explorer* interface.

The STSIMULATOR tool was designed and implemented by Lars Frantzen.

Ta2Torx The TA2TORX tool provides access to specifications given as network of Timed Automata. We discuss this in greater detail in Chapter 6.

The TA2TORX tool was designed and implemented by Henrik Bohnenkamp.

SPEX The SPEX tool [vY07] (Simple Promela EXplorer for TorX) provides access to specifications given in PROMELA [Hol91]. A PROMELA specification typically describes a system (with possible internal communication within the system) with its interactions with its environment. However, PROMELA makes no distinction between communication (interactions) within a system, and the interactions between the system and its environment. For our testing, we are not interested in the former, but we need the latter: we need a description of the system in terms of its interactions with its environment[3]. Therefore, SPEX implements two extensions to the Promela language. The first is an IO extension to open the specified system for testing, by adding a mechanism to describe the communication/interaction between the system and its environment. The

[3]TROJKA [dT00], a predecessor of SPEX, solved this by extending PROMELA with a keyword that allows the specifier to indicate that particular communication is *observable*; TROJKA then makes these available to the testing tool, and turns the other ones into internal (τ) actions.

second is a Set extension to close such system description for verification, by replacing input actions by symbolic variables.

The SPEX tool was designed and implemented Jeroen van Yperen.

Because all PROMELA statements, except the interactions between system and environment, are mapped onto τ-transitions, an LTS thus obtained typically contains long "chains" of τ transitions, where the states on such chain only have one incoming and one outgoing transition, both labelled with τ. An obvious optimisation is to replace such chain $p \xrightarrow{\tau} p' \xrightarrow{\tau} p'' \ldots \xrightarrow{\tau} q$ by a single transition $p \xrightarrow{\tau} q$, and to ignore the intermediate states (p', p'', etc.). We experimented with this optimisation when we experimented with SPEX.

CADP CADP [CAD] ("Construction and Analysis of Distributed Processes", formerly known as "CAESAR/ALDEBARAN Development Package") is a toolbox for the design of asynchronous concurrent systems, such as communication protocols, distributed systems, asynchronous circuits, multiprocessor architectures, web services, etc. It offers compilers for high-level descriptions written in a.o. LOTOS [ISO89]. It also offers the BCG (binary coded graphs) file format, which allows storing of large graphs.

CADP offers access to the languages and file formats that it supports via (a.o.) the OPEN/CAESAR programming interface, an interface that provides (more than) the functionality needed for our Explorer interface. A *torx-explorer* interface has been constructed to access specifications via the OPEN/CAESAR interface.

LTSmin LTSMIN [LTS, BvdPW09] is a toolset for model checking and manipulating labelled transition systems. LTSMIN already connects a sizeable number of existing (verification) tools: muCRL, mCRL2, DiVinE, SPIN via an included version of SpinJa, NIPS, CADP and opaal. Moreover, it allows to reuse existing tools with new state space generation techniques.

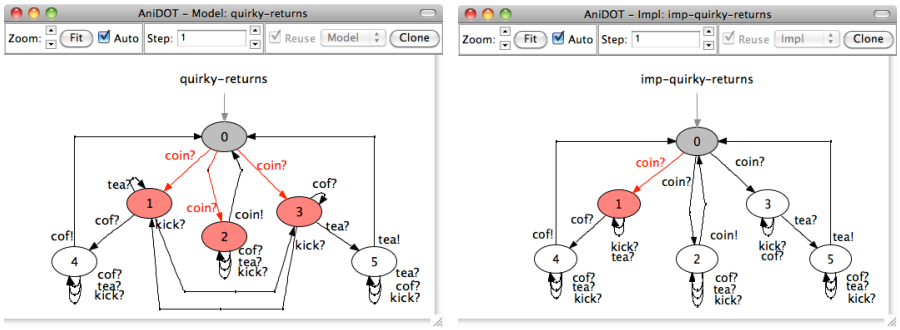
The LTSMIN toolset contains tools to access e.g. mCRL2 models using the *torx-explorer* interface.

mCRL2 mCRL2 [mCR, GKM+08] is a formal specification language with an associated toolset. The toolset can be used for modelling, validation and verification of concurrent systems and protocols.

The toolset supports a collection of tools for linearisation, simulation, state-space exploration and generation and tools to optimise and analyse specifications. Moreover, state spaces can be manipulated, visualised and analysed.

The MCRL2 toolset contains the LPS2TORX tool to access muCRL and mCRL2 models using the *torx-explorer* interface.

Bibliographical note Some of the text describing the tools, toolsets and languages has been taken from the web-sites describing the tools.



(a) (b)

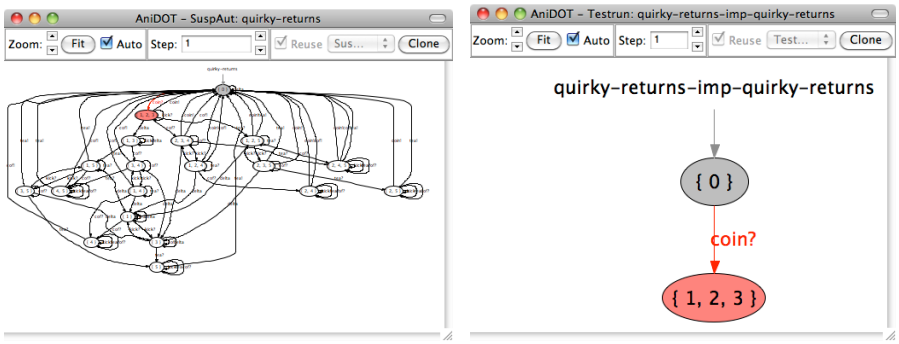
Figure 4.7: Animation of the visualisation of the Quirky Coffee Machine specification (a), and of the Kick-Insensitive implementation (b), after the first test step in which stimulus ?coin was applied. In both (a) and (b) the grey highlights are the same: state 0, to indicate that that was the previous state, and the initial transition, to show how we "reached" state 0. The difference is in the red highlights. In (a) states 1, 2, and 3 are highlighted in red, as are the ?coin transitions that lead to these states from state 0. In (b) only state 1 is highlighted in red, as is (only) the ?coin transition that leads to this state from state 0.

4.2.5 Visualisation

To fulfil requirement 14—the tool should provide insight in the theory and algorithms that it implements, e.g. by visualisation—we let our tool visualise the specification, the suspension automaton (SA), and, when we use a simulated model as SUT, also the implementation model, and we animate these during test runs. (During a test run, we also visualise the test steps that are executed in a dynamically updated message sequence chart, but we do not discuss that here further.)

Animation entails highlighting the visual elements that correspond with the LTS resp. SA states that correspond with the current test step, and the transition(s) that brought us there. This is shown in Figure 4.7 (a) and (b), and in Figure 4.8 (a). We also can show a view of the SA that is constructed during the test run: it shows only the pseudo-states and transitions that we have traversed so far (during the run); see Figure 4.8 (b). Therefore, this latter view is much smaller, and thus, more understandable, than the visualisation of the entire SA. In both views of the SA, the node-labels show the set of LTS labels that belong to the pseudo-state.

The animations shown in Figures 4.7 and 4.8 are of a test run with the quirky coffee machine and the kick-insensitive implementation, after the first test step in which stimulus ?coin was applied. The animations in Fig. 4.7 (a) and (b) show clearly the non-determinism that is present in the specification (states 1, 2, and 3 are highlighted, as are all ?coin transitions that lead to these states), but absent (i.e., resolved, by making a random choice among the enabled ?coin



(a) (b)

Figure 4.8: Animation of suspension automaton (SA) of the Quirky Coffee Machine specification (a), and of the states and transitions of the SA traversed so far (b).

transitions) in the implementation (only state 1 is highlighted, as is only one transition: the ?coin transition that leads to state 1). It is also clearly visible that in this case[4] the implementation has chosen the branch in which it only serves coffee.

Below we discuss how the DerivationEngine supports visualisation and animation.

Supporting visualisation For the visualisation of the specification, we either, when available (like for JARARACA and STSIMULATOR), use a DOT representation of the specification, specifically created for visualisation; otherwise, we use a visualisation of the state space of the LTS of the specification. In the latter case, we just generate the state space of the specification: our Explorer interface allows this, and this works fine as long as the state space is small enough for this to finish in a time that is acceptable to the user.

Note that (node, edge) position information is optional: when we have position information (which we have in the case of GRAPHML models), we use it for the layout; otherwise, we let the DOT tool of the GRAPHVIZ toolkit compute the layout (this is done, for example, for both visualizations shown in Figure 4.8).

Supporting animation For the animation, we need to be able to highlight (visual elements that correspond to)

1. the LTS states that belong to the current pseudo-state, and
2. the LTS transitions that lead from the previous pseudo-state to the current one. (At the start of the test run, we highlight the initial transition.)

For this we need access to states and transitions of the LTS, as we discuss below.

[4]Random number seed 1012308167.

Ad 1: access to LTS states In our earlier discussion of the pseudo-state type, we did provide it with a method m() that provides access to the LTS states that belong to it. We refine this, by adding methods that give separate access to (1) the LTS states that are directly reached (m_d), and (2) those that are indirectly reached via one or more τ transitions ($unfoldResult.m_e$). This allows us to color these with different colors in the animation, to better illustrate the work done by *unfold*.

Ad 2: access to LTS transitions To be able to highlight the LTS transitions, we let the Primer store the following:

1. With each pseudo-state, we store the τ-transitions that "belong" to it, i.e. those that are traversed by method *unfold*.
2. With each pseudo-state, we store a mapping from the enabled input and output labels to the corresponding set of LTS transitions. (We store such set in a pseudo-transition type.)

Note that the information stored in *destmap* does not suffice, because, to be able to highlight the transitions, we need to know not only the destination LTS state and the transition label, but also the source LTS state. The *destmap* does provide the destination LTS state, but it does not provide access to the source LTS state.

4.2.6 Algorithm for uioco

In Section 1.2.3 we mentioned that we will validate requirement 10—it should be easy to incorporate new conformance relations—by discussing the addition of support for implementation relation **uioco**. Here we discuss the changes w.r.t. Algorithm 4.1 that are necessary to support the **uioco** relation.

As mentioned in Section 2.2.5, **uioco** was designed to better handle specifications that are not input-enabled. Relations **ioco** and **uioco** differ in how the set of enabled inputs is computed, and this difference only shows in case of non-determinism, i.e., when the tester state consists of a number of LTS states (say, set P). With **ioco** the *union* of the sets of enabled inputs of *all* the individual LTS states in P is used; with **uioco** the *intersection* is used, of the sets of enabled inputs of *stable* states in P (i.e. of those states in P that have no outgoing τ-transitions).

In the algorithm for **ioco**, when we encounter an input transition, we immediately add the input label to *inputs*, the set of enabled inputs, and update *destmap*, the mapping from labels to (destination) LTS state set (see lines 13 and 14 of Algorithm 4.1). However, for **uioco**, we do not know whether we even should consider the inputs of an LTS state, until we know whether the state is stable—this is when we have dealt with all outgoing transitions of the state. And even at that moment, we can only use those inputs, enabled in the state, to update the intermediate result of the intersection of enabled inputs. Only when we have dealt with (all transitions of) all states we know which inputs are enabled, and thus, only then we update *destmap*.

Details In Algorithm 4.2 we show the changes that are necessary w.r.t. Algorithm 4.1. In lines 4–5 we show the changes to the global variables: we initialise *inputs* to $\perp$ (instead of to $\emptyset$, so we can distinguish "uninitialised" from "empty"), and we add *inputTransitions*, which will hold all transitions with an input label that we encounter.

In lines 9–10 we added local variables (re-initialised for each state that we handle) *isStable* and *stateInputs*. Variable *isStable*, initialised to **true**, is set to **false** as soon as a transition with a τ label is encountered. Variable *stateInputs* collects the inputs that are enabled for the state that is being handled. When all transitions of a state have been handled, these two variables are used to update *inputs*, the set of enabled inputs (lines 21–24).

Finally, when all states have been handled, *inputs* is initialised if it had not been initialised before, i.e. when there were no transitions (lines 26–27), and *destmap* is updated using the value of *inputs* (lines 28–30).

4.2.7 Interpreting divergent states as quiescent

The Primer algorithm that we gave in Section 4.2.3 uses the "avoid looping" approach of Section 4.1.2 to deal with τ-cycles. We now discuss the two "divergent states are quiescent" approaches of Section 4.1.3 and Section 4.1.4. These either just add a δ-self-loop to divergent states (see Algorithm 4.3) or create copies of divergent states (see Algorithm 4.4). In both approaches the following two activities can be identified:

1. identifying the divergent states—this we do first, and this is followed by
2. marking the divergent states as being quiescent, by adding a δ-self-loop, resp. by creating copies.

For each of these two activities, we discuss how they are performed. Algorithm 4.3 shows the changes w.r.t. Algorithm 4.1, to identify the divergent states and add a τ-loop to them, and Algorithm 4.4 shows the changes w.r.t. Algorithm 4.3, to create copies of divergent states.

Ad 1: Identifying divergent states To identify the divergent states we adapted the path-based strong component algorithm [Gab00] to our needs. For strong components that consist of more than one state, all states are divergent. For a single-state strong component, its state is divergent when it has a τ-self-loop. Therefore, while computing the strong components, we remember which states have a τ-self-loop. There are two important constraints on our adaptation of the algorithm, due to the context in which we use it:

- we choose to use an iterative version, to reduce the "distance" from Algorithm 4.1 (whereas the strong component algorithm is defined using recursion);
- while we work on a state, we choose to handle all its outgoing transitions, before we start working on (the transitions of) another state (we do this, because the Explorer gives us all transitions of a state together).

As in Algorithm 4.1, when working on a state p we process all its outgoing transitions, before we select the next state to work on. As a consequence, any "new" (to be processed) states that are found while working on a state have to

Algorithm 4.2: *P.unfold*() for **uioco** Primer — changes w.r.t. Algo. 4.1

input : P, a pseudo-state
e, an Explorer that gives access to the Specification
o, an IO-Oracle
output: P, the pseudo-state, unfolded if it wasn't already unfolded

```
1  begin
2      if P.unfoldResult = ⊥ then
3          ⋮
4          inputs ⟵ ⊥
5          inputTransitions ⟵ ∅
6          ⋮
7          while work ≠ ∅ do
8              ⋮
9              isStable ⟵ true
10             stateInputs ⟵ ∅
11             foreach ⟨l, p'⟩ ∈ e.menu(p) do
12                 kind ⟵ o.kind(l)
13                 if kind = I then
14                     stateInputs ⟵ stateInputs ∪ {l}
15                     inputTransitions ⟵ inputTransitions ∪ {⟨p, l, p'⟩}
16                 ⋮
17                 else if kind = τ then
18                     isStable ⟵ false
19                 ⋮
20             ⋮
21             if isStable and inputs = ⊥ then
22                 inputs ⟵ stateInputs
23             else if isStable then
24                 inputs ⟵ inputs ∩ stateInputs
25         ⋮
26         if inputs = ⊥ then
27             inputs ⟵ ∅
28         foreach ⟨p, l, p'⟩ ∈ inputTransitions do
29             if l ∈ inputs then
30                 destmap[l] ⟵ destmap[l] ∪ {p'}
31         P.unfoldResult ⟵ ⟨inputs, outputs, destmap, extra⟩
32     return P
```

Algorithm 4.3: *P.unfold*() for τ-loop detection — changes w.r.t. Algo. 4.1

```
1  begin
2      if P.unfoldResult = ⊥ then
3          wStack ⟵ pStack ⟵ sStack ⟵ empty; c ⟵ 0
4          hasSCC ⟵ hasTauSelfLoop ⟵ ∅; pre ⟵ {p → ⊥ | p ∈ S}
5          if m_d ≠ ∅ then
6              wStack.push(⟨⊥, m_d⟩)
7          ⋮
8          while |wStack| > 0 do
9              pick state p ∈ wStack.top().set
10             wStack.top().set ⟵ wStack.top().set \ {p}
11             if pre[p] = ⊥ then
12                 pre[p] ⟵ c; increment c
13                 pStack.push(p); sStack.push(p)
14                 isQuiescent ⟵ true; tauChildren ⟵ ∅
15                 foreach ⟨l, p'⟩ ∈ e.menu(p) do
16                     ⋮
17                     else if kind = τ then
18                         isQuiescent ⟵ false
19                         if p' ∉ P.m_d ∧ p' ∉ extra then
20                             extra ⟵ extra ∪ {p'}
21                         if p = p' then
22                             hasTauSelfLoop ⟵ hasTauSelfLoop ∪ {p}
23                         if pre[p'] = ⊥ then
24                             tauChildren ⟵ tauChildren ∪ {p'}
25                         else if p' ∉ hasSCC then
26                             while pre[pStack.top()] > pre[p'] do
27                                 pStack.pop()
28                 ⋮
29                 wStack.push(⟨p, tauChildren⟩)
30             while |wStack| > 0 and |wStack.top().set| = 0 do
31                 if |pStack| > 0 and wStack.top().pnt = pStack.top() then
32                     S ⟵ sStack.popUptoInc(wStack.top().pnt)
33                     hasSCC ⟵ hasSCC ∪ S
34                     if |S| > 1 or ∀s ∈ S : s ∈ hasTauSelfLoop then
35                         outputs ⟵ outputs ∪ {⟨δ, ⊥⟩}
36                         destmap[δ] ⟵ destmap[δ] ∪ S
37                     pStack.pop()
38                 wStack.pop()
39         ⋮
40     return P
```

Algorithm 4.4: *P.unfold*() for making copies of divergent states — changes w.r.t. Algo. 4.3

```
1  begin
2      if P.unfoldResult = ⊥ then
3          ⋮
4          while |wStack| > 0 do
5              ⋮
6              while |wStack| > 0 and |wStack.top().set| = 0 do
7                  if |pStack| > 0 and wStack.top().pnt = pStack.top() then
8                      ⋮
9                      if |S| > 1 or ∀s ∈ S : s ∈ hasTauSelfLoop then
10                         foreach q ∈ S do
11                             create input-only clone q_qos of q
12                             destmap[δ] ⟵ destmap[δ] ∪ {q_qos}
13                     ⋮
14                 ⋮
15             ⋮
16     return P
```

be remembered, such that they can be processed later. The two main differences between Algorithm 4.1 and Algorithm 4.3 are about (a) how we store the states that we still have to work on, and (b) how we select the next state to work on.

In Algorithm 4.1 we store the states that we have to work on in a set *work*. When we encounter a state that we have not seen before, we just add it to the set (line 23). We process the states in *work* in an unspecified order: at line 7 we only state "pick state $p \in$ *work*".

In Algorithm 4.3, we store unprocessed states on a stack of sets of states, where we group siblings (states that are children of the same state, i.e. reached from the same state via a τ-transition) together in a set. With each such set we also store the parent state—we need the parent for the computations that are done after all outgoing transitions have been processed (including the work that in the original algorithm is done in recursive calls). To stress the difference with the set *work*, we refer to it as *wStack*. We use *wStack*.top().set to refer to the set on the top of *wStack*, and *wStack*.top().pnt to refer to the associated parent. We process the states in the sets on *wStack* such, that we process children before we process siblings. While processing the outgoing transitions of a state p, we collect all its τ-reached "children" that are still unprocessed in a set *tauChildren* (line 24). When we have seen all outgoing transitions of p, we push a tuple $\langle p, \mathit{tauChildren} \rangle$ onto *wStack* (line 29). When we select the next

state to work on, we pick an arbitrary one from the set of the topmost tuple on *wStack* (line 9).

To find the strong components (strongly connected component, SCC), we use two additional stacks, *pStack*, and *sStack*, a map *pre*, a set *hasSCC* and a counter *c*. In addition, we use set *hasTauSelfLoop* to remember those states that have a τ-self-loop. Onto *sStack* we collect τ-connected states that have not yet been assigned to a SCC, in the order in which they are reached by depth-first search. On *pStack* we have states for which it has not yet been decided whether they belong to different SCCs. Map *pre* associates with each state a pre-order visit number. Set *hasSCC* contains those states that have been assigned to an SCC. Counter *c* counts the number of states that we have seen. On these data structures we use one special operation: method *sStack*.popUptoInc(p) (line 32) pops-and-returns state p, and all states on top of p, from *sStack*.

To make the SCC finding approach work, we have to do a depth-first search, and that is why, while unfolding unprocessed states, we unfold children before we unfold siblings. When we need a state to process, we take one (p) from the set at the top of *wStack*—there should be one, because at the start of the algorithm, we only push set m_d onto *wStack* if it is non-empty, and in the last part of the algorithm (in lines 30–38) we make sure that we pop all topmost *wStack* items that have an empty set. While processing p, we first check whether it already has a pre-order visit number associated with it. If so, it has been processed already, and we don't process it again. Otherwise, we associate the next visit number with it (and increment *c*), and push state p to both *sStack* and *pStack*. Then, we process the outgoing transitions of p.

We process non-τ-transitions as in Algorithm 4.1. When processing a τ-transition, we first check whether it is a self-loop, and if so, update *hasTauSelfLoop*. Then, we check whether the destination state p' is still unprocessed. If so, we add it to *tauChildren*. Otherwise, if p' has not yet been assigned to an SCC, we pop from *pStack* those states that are on the same SCC as p'. When we have processed all outgoing transitions of p, we push a tuple $\langle p, \mathit{tauChildren}\rangle$ onto *wStack*.

Then, we check whether the (children) set at the top of the *wStack* is empty. If so, we have finished the recursive processing of the children of the associated parent (so we will pop *wStack*) and we must check whether we have identified (completed) an SCC—if so, we pop it from *sStack*, and we pop *pStack*. We then check whether the state(s) on the SCC are divergent, and if so, act accordingly (lines 34–36).

We now first discuss activity 2: marking the divergent states as quiescent, and then we give an example to illustrate Algorithm 4.3.

Ad 2: Marking the divergent states as quiescent In Algorithm 4.3 we show the approach where we mark divergent states by adding δ-self-loops to them, i.e. where we treat divergent states identical to quiescent states. We do not show treatment of quiescent states (but see lines 25–26 of Algorithm 4.1); we treat divergent states in lines 35–36.

In Algorithm 4.4 we show the approach where we mark divergent states by creating copies of the divergent states. The copies are made at line 11. For each

copy q_{qos} of state q it should hold that $\forall a \in L_I :$ if $q \xrightarrow{a} q'$ then $q_{qos} \xrightarrow{a} q'$, and q_{qos} should have no other outgoing transitions.

Example: Identifying divergent states

We illustrate Algorithm 4.3 with the self-kicking coffee machine of Figure 4.1. We look at two invocations of the algorithm:

(a) for the initial state (i.e. for the DerivationEngine interface start method,
(b) for the state reached from the initial state by doing a ?coin transition (i.e. s **after** ?coin).

For each invocation we describe the main steps of the algorithm; we show the effect on the main data structures in Table 4.7 resp. Table 4.8.

Ad a: initial state At the start of this invocation of *unfold*, the pseudo-state for which we invoke *unfold* contains $\langle\{0\}, \bot\rangle$, i.e. $m_d = \{0\}$ (i.e. it holds only the initial state of the model), and *unfoldResult* $= \bot$. Thus, we start by pushing $\langle\bot, \{0\}\rangle$ onto *wStack*, and intializing the other variables. Then, we enter the main while loop at line 8 (*wStack* contains one element). We take state $p = 0$ from the set in *wStack* (thus, now *wStack* contains $\langle\bot, \emptyset\rangle$). No pre-order visit number has been associated with state 0 so far, so we enter the if-clause at line 11. We associate number 0 with state 0, increment counter c, push state 0 on both *sStack* and *pStack*, initialize *isQuiescent* and *tauChildren*, obtain the outgoing transitions from state 0, and start processing them (line 15).

wStack	p	l	p'	*tauChildren*	*sStack*	*pStack*	SCC
$\langle\bot, \{\emptyset\}\rangle$							
$\langle\bot, \emptyset\rangle$	0						
$\langle\bot, \emptyset\rangle$	0				0		
$\langle\bot, \emptyset\rangle$	0				0	0	
$\langle\bot, \emptyset\rangle$	0	?coin	1		0	0	
$\langle\bot, \emptyset\rangle$	0	?coin	2		0	0	
$\langle\bot, \emptyset\rangle$	0	?coin	3		0	0	
$\langle\bot, \emptyset\rangle$	0			$\emptyset$	0	0	
$\langle\bot, \emptyset\rangle$ $\langle 0, \emptyset\rangle$	0				0	0	
$\langle\bot, \emptyset\rangle$ $\langle 0, \emptyset\rangle$	0					0	$\{0\}$
$\langle\bot, \emptyset\rangle$ $\langle 0, \emptyset\rangle$	0					0	
$\langle\bot, \emptyset\rangle$ $\langle 0, \emptyset\rangle$	0						
$\langle\bot, \emptyset\rangle$							

Table 4.7: Changes to main data structures of Algorithm 4.3 while unfolding the pseudo-state that holds the initial state of the self-kicking coffee machine of Fig. 4.1. The horizontal lines separate phases of the algorithm (initialization, processing of transitions, popping of stacks and detection of SCC). Column SCC shows the strong component that is found.

State 0 has three outgoing transitions: $\xrightarrow{?\mathsf{coin}} 1$, $\xrightarrow{?\mathsf{coin}} 2$, and $\xrightarrow{?\mathsf{coin}} 3$. After processing these transitions $inputs = \{?\mathsf{coin}\}$ and $destmap[?\mathsf{coin}] = \{1, 2, 3\}$; other variables are unchanged, i.e. $isQuiescent = \textbf{true}$, $tauChildren = \emptyset$. Thus, we mark state 0 as quiescent (lines 25–26 of Algorithm 4.1), and we push $\langle 0, \emptyset \rangle$ onto *wStack*.

Now, we enter the while loop at line 30. The pnt field of the tuple that we just pushed equals the top element of *pStack* (line 31), i.e. we have found a strong component, which we obtain by pushing the topmost elements of *sStack*—in this case just the single state 0. This state is not in *hasTauSelfLoop*, i.e. it is not divergent, so we need not mark it as quiescent. We pop *pStack* and *wStack*. Now *wStack* contains $\langle \bot, \emptyset \rangle$, so we enter the loop at line 30 once more, but this time we only pop *wStack*.

Now, *wStack* is empty, so, the only things left to do, are to set field *unfoldResult* (line 27 of Algorithm 4.1), and to return.

Ad b: state reached by ?coin **from initial state** At the start of this invocation of *unfold*, the pseudo-state contains $\langle \{1, 2, 3\}, \bot \rangle$, i.e. $m_d = \{1, 2, 3\}$ and $unfoldResult = \bot$. Again, we start by initializing; now $wStack = \langle \bot, \{1, 2, 3\} \rangle$.

Again, we enter the main while loop (line 8). We pick a state p from the set at the top of *wStack*; assume we pick 1. We associate a pre-order visit number (0) with this state; we push the state onto *sStack* and *pStack*; and we obtain its outgoing transitions ($\xrightarrow{?\mathsf{tea}} 1$, $\xrightarrow{?\mathsf{cof}} 4$, and $\xrightarrow{\tau} 3$) and process them.

After processing the two input transitions we have $inputs = \{?\mathsf{tea}, ?\mathsf{cof}\}$ and $destmap[?\mathsf{tea}] = \{1\}$, $destmap[?\mathsf{cof}] = \{4\}$. The τ-transition is no self-loop, and its destination is "new", and thus added to *tauChildren*. After processing these transitions, we push $\langle 1, \{3\} \rangle$ onto *wStack*, which already contains $\langle \bot, \{2, 3\} \rangle$. The test at line 30 fails, so we loop in the main loop.

We pick a state from the set at the top of *wStack*; then $p = 3$ and $wStack = (\langle \bot, \{2, 3\} \rangle, \langle 1, \emptyset \rangle)$. We have not processed state 3 before, so we associate a pre-order visit number (1) with it, and push it onto *sStack* and *pStack* (both already contain state 1). We obtain its outgoing transitions ($\xrightarrow{?\mathsf{cof}} 3$, $\xrightarrow{?\mathsf{tea}} 5$, $\xrightarrow{\tau} 1$), and process them.

After processing the two input transitions we have $inputs = \{?\mathsf{tea}, ?\mathsf{cof}\}$ and $destmap[?\mathsf{tea}] = \{1, 5\}$, $destmap[?\mathsf{cof}] = \{4, 3\}$. Again, the τ-transition is no self-loop; this time its destination is neither "new", nor part of an SCC, and thus we pop elements from *pStack* as given in lines 26–27. Because $pre[pStack.\text{top}()] = pre[3] = 1 > pre[p'] = pre[1] = 0$ we pop the topmost item from *pStack*; we do not pop the other item, because now $pre[pStack.\text{top}()] = pre[1] = 0 \not> pre[p'] = pre[1] = 0$. After processing these transitions, we push $\langle 3, \emptyset \rangle$ onto *wStack*; it now contains $(\langle \bot, \{2, 3\} \rangle, \langle 1, \emptyset \rangle, \langle 3, \emptyset \rangle)$; $sStack = (1, 3)$; $pStack = (1)$.

The test at line 30 now holds. For the top element of *wStack*, the test at line 31 does not hold, so we pop that top element; $wStack = (\langle \bot, \{2, 3\} \rangle, \langle 1, \emptyset \rangle)$. The test at line 30 still holds, and moreover, also the test at line 31 holds. Thus, we have found an SCC, that contains state 1, and all elements on top of that state on *sStack*, i.e. the SCC is $\{3, 1\}$, and we pop these states from *sStack*. The SCC is divergent (contains more than one element), so we update *outputs*

wStack	p	l	p'	*tauChildren*	*sStack*	*pStack*	SCC
$\langle \bot, \{1,2,3\}\rangle$							
$\langle \bot, \{2,3\}\rangle$	1						
$\langle \bot, \{2,3\}\rangle$	1				1		
$\langle \bot, \{2,3\}\rangle$	1				1	1	
$\langle \bot, \{2,3\}\rangle$	1	?tea	1		1	1	
$\langle \bot, \{2,3\}\rangle$	1	?cof	4		1	1	
$\langle \bot, \{2,3\}\rangle$	1	τ	3		1	1	
$\langle \bot, \{2,3\}\rangle$	1			$\{3\}$	1	1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \{3\}\rangle$	1				1	1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	3				1	1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	3				1 3	1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	3				1 3	1 3	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	3	?cof	3		1 3	1 3	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	3	?tea	5		1 3	1 3	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	3	τ	1		1 3	1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	3			$\emptyset$	1 3	1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle\ \langle 3, \emptyset\rangle$	3				1 3	1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	1				1 3	1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	1					1	$\{1,3\}$
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	1					1	
$\langle \bot, \{2,3\}\rangle\ \langle 1, \emptyset\rangle$	1						
$\langle \bot, \{2,3\}\rangle$							
$\langle \bot, \{3\}\rangle$	2						
$\langle \bot, \{3\}\rangle$	2				2		
$\langle \bot, \{3\}\rangle$	2				2	2	
$\langle \bot, \{3\}\rangle$	2	?cof	2		2	2	
$\langle \bot, \{3\}\rangle$	2	?tea	2		2	2	
$\langle \bot, \{3\}\rangle$	2	!coin	0		2	2	
$\langle \bot, \{3\}\rangle$	2	τ	2		2	2	
$\langle \bot, \{3\}\rangle$	2			$\emptyset$	2	2	
$\langle \bot, \{3\}\rangle\ \langle 2, \emptyset\rangle$	2				2	2	
$\langle \bot, \{3\}\rangle\ \langle 2, \emptyset\rangle$	2					2	$\{2\}$
$\langle \bot, \{3\}\rangle\ \langle 2, \emptyset\rangle$	2					2	
$\langle \bot, \{3\}\rangle\ \langle 2, \emptyset\rangle$	2						
$\langle \bot, \{3\}\rangle$							
$\langle \bot, \emptyset\rangle$	3						
$\langle \bot, \emptyset\rangle$							

Table 4.8: Changes to main data structures of Algo. 4.3 while unfolding the state reached by ?coin from the initial state of the self-kicking coffee machine of Fig. 4.1. (Although in the algorithm we do not change p when we pop *wStack*, here we show p changed to the parent on *wStack*.) Horizontal lines separate phases of the algorithm; column SCC shows strong components that are found.

and *destmap*: $outputs = \{\langle\delta, \bot\rangle\}$ and $destmap[\delta] = \{1, 3\}$. We also pop *pStack* and *wStack*; now $wStack = (\langle\bot, \{2, 3\}\rangle)$, and the test at line 30 fails, so we loop in the main loop.

We pick a state from the set at the top of *wStack*; assume we pick 2, then $p = 2$ and $wStack = (\langle\bot, \{3\}\rangle)$. We have not processed state 2 before, so we associate a pre-order visit number (2) with it, and push it onto *sStack* and *pStack* (both were empty). We obtain its outgoing transitions ($\xrightarrow{?\mathsf{cof}} 2$, $\xrightarrow{?\mathsf{tea}} 2$, $\xrightarrow{!\mathsf{coin}} 0$, $\xrightarrow{\tau} 2$), and process them. After processing the two input transitions and the output transition we have $inputs = \{?\mathsf{tea}, ?\mathsf{cof}\}$, $outputs = \{\delta, !\mathsf{coin}\}$ and $destmap[?\mathsf{tea}] = \{1, 5, 2\}$, $destmap[?\mathsf{cof}] = \{4, 3, 2\}$, $destmap[\delta] = \{1, 3\}$ and $destmap[!\mathsf{coin}] = \{0\}$. Now, the τ-transition is a self-loop, so we add state 2 to *hasTauSelfLoop*. Moreover, $pre[p'] = pre[2] = 2 \neq \bot$, and state 2 is not yet part of an SCC, so we test whether we should pop *pStack*, and this test fails, so *pStack* is not changed. After processing these transitions, we push $\langle 2, \emptyset\rangle$ onto *wStack*; it now contains $(\langle\bot, \{3\}\rangle, \langle 2, \emptyset\rangle)$; $sStack = (2)$; $pStack = (2)$.

The test at line 30 now holds. For the top element of *wStack*, the test at line 31 also holds. Thus, we have found an SCC; this time it contains only state 2; we pop that state from *sStack*. Even though the SCC consists of a single state, that state is divergent (because it is in *hasTauSelfLoop*), so we update (*outputs* and) *destmap*: $destmap[\delta] = \{1, 3, 2\}$. We pop *pStack* and *wStack*; now $wStack = (\langle\bot\{3\}\rangle)$. The test at line 30 fails, so we loop in the main loop.

We pick the last element from the set at the top of *wStack*; thus we pick 3, then $p = 3$ and $wStack = (\langle\bot, \emptyset\rangle)$. We have seen state 3 before, so the test at line 11 fails. The test at line 30 holds, but the test at line 31 fails. Thus, we only pop *wStack*, which is now empty, and thus the condition for the main loop fails. All that is left to do is setting the *unfoldResult* field, and returning.

4.3 DerivationEngine for Guided Testing

We now discuss the DerivationEngine for guided test derivation, i.e. the DerivationEngine that gives access to both a specification and an observation objective.

Our design of this DerivationEngine aims to achieve the following:

1. support the "guided mode" part of requirement 7 (support random mode and guided mode), by extending our decomposition of Figure 4.4 with components to access the guidance information (test purpose, observation objective), and to compute the cross product between the specification and the guidance information.

4.3.1 Components

The extended decomposition is depicted in Figure 4.9. It extends the decomposition of Figure 4.4 with 3 components: (1) a second Explorer, which provides access to the observation objective; (2) a *traces* Primer, which provides access to a determinized "version" of the observation objective; (3) a Combinator, which computes the cross-product between specification and observation objective.

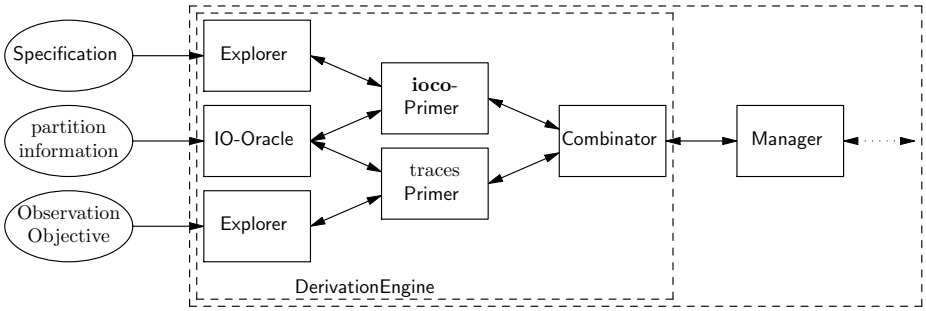


Figure 4.9: Decomposition of DerivationEngine extended to handle guidance.

Moreover, because the observation objective may contain labels that represent quiescence, both the Explorer and IO-Oracle are extended to deal with these.

4.3.2 Interfaces

Between (traces) Primer and Explorer: the Explorer interface

1. start $:\rightarrow S$
2. menu : $S \rightarrow \mathcal{P}((L_\tau \cup \{\delta\}) \times S)$

Table 4.9: Signature of Explorer interface functions, which can return a label that represents quiescence.

The return type of the Explorer function menu is extended, to allow it to return labels that represent quiescence, i.e. δ. This is the only change to the Explorer interface.

Between Primer and IO-Oracle: the IO-Oracle interface

There may be labels in an observation objective that represent quiescence. As mentioned above, we extend the IO-Oracle interface with the ability to identify such labels. We assume that the "partitioning information", has been extended as well, and that an initialised IO-Oracle has (in addition to functions isI, isU, and isH) a function isQ that tests for quiescence.

1. kind : $L_\tau^\delta \times \rightarrow \{\text{I}, \text{U}, \tau, \delta\}$

Table 4.10: Signature of IO-Oracle interface function that supports quiescence (L_τ^δ represents $L_\tau \cup \{\delta\}$).

Algorithm 4.5: *P.unfold*() for *traces* Primer — changes w.r.t. Algo. 4.1

```
   input  : P, a pseudo-state
            e, an Explorer that gives access to the observation objective
            o, an IO-Oracle
   output: P, the pseudo-state, unfolded if it wasn't already unfolded
 1 begin
 2   if P.unfoldResult = ⊥ then
 3     ⋮
 4     while work ≠ ∅ do
 5       ⋮
 6       foreach ⟨l, p'⟩ ∈ e.menu(p) do
 7         kind ⟵ o.kind(l)
 8         ⋮
 9         else if kind = δ then
10           outputs ⟵ outputs ∪ {⟨δ, ⊥⟩}
11           destmap[δ] ⟵ destmap[δ] ∪ {p'}
12     P.unfoldResult ⟵ ⟨inputs, outputs, destmap, extra⟩
13   return P
```

Ad 1: kind Function kind tells to what partition a given label belongs. It is defined as follows (with $l \in L_\tau \cup \{\delta\}$):

$$\mathsf{kind}(l, \langle isI, isU, isH, isQ\rangle) \;=_{\text{def}}\; \begin{cases} \text{I} & \text{if } isI(l) \\ \text{U} & \text{if } isU(l) \\ \tau & \text{if } isH(l) \\ \delta & \text{if } isQ(l) \end{cases} \tag{4.27}$$

Between (ioco or traces) Primer and Combinator: the Primer interface

The Primer interface is unchanged w.r.t. random on-line testing, as is the implementation of this interface in the **ioco** Primer. However, the *traces* Primer has (compared to the **ioco** Primer) a slightly different implementation of this interface. The *traces* Primer does not synthesise δ labels, but when it receives such a label from its Explorer, it passes a δ label on. The changes are isolated to the *unfold*() function. Algorithm 4.5 shows the changes w.r.t. Algorithm 4.1: there is an additional case, to pass on δ labels (lines 9–11); moreover, variable *isQuiescent* is no longer set nor used.

Provided by the Combinator: the DerivationEngine interface

In the guided case, it is the Combinator that is the component that realises the interface that the DerivationEngine provides to its user, i.e. now the Combinator

interface coincides with the DerivationEngine interface (see Fig. 4.9). The signature of this interface was given in Table 3.8. In Section 4.3.3 below we discuss how the interface is realised.

4.3.3 Combinator algorithm

We now discuss how the Combinator can provide the DerivationEngine interface using the functionality offered by **ioco** Primer and *traces* Primer.

To realise the interface, we once more refine the pseudo-state type (introduced in the previous chapter (see Table 3.7), to obtain the pseudo-state type that is used by the Combinator. The internal structure and interface of this pseudo-state type differ from those of the pseudo-state type that we used in the Primer for random testing. In particular, we added two utility methods: *resultTuple*(), and $v()$, and we changed the return type of method $P.n(l)$. The resulting pseudo-state type is shown in Table 4.11, and discussed below.

Signature:	
PS :	$(P_S \uplus \bot) \times (P_G \uplus \bot) \to P_C$
m :	$P_C \to P_S \uplus \bot$
g :	$P_C \to P_G \uplus \bot$
resultTuple :	$P_C \to (P_C \uplus \bot) \times (L_V \uplus \bot)$
i :	$P_C \to \mathcal{P}(L_I)$
o :	$P_C \to \mathcal{P}(L_U^\delta \times (L_V \uplus \{\bot\}))$
v :	$P_C \to L_V \uplus \bot$
n :	$P_C \times L^\delta \to (P_C \uplus \bot) \times (L_V \uplus \bot)$
unfold :	$P_C \to P_C$
Definition:	
PS(p_s, p_g)	$= \langle p_s, p_g, \bot \rangle$ (a new pseudo-state)
p_c.m()	$= p_c.m$
p_c.g()	$= p_c.g$
p_c.*resultTuple*()	$= \langle p_c, p_c.v() \rangle$
$p_c.i()$	$= p_c.unfold().unfoldResult.i$
$p_c.o()$	$= p_c.unfold().unfoldResult.o$
$p_c.hasOutputs()$	$= p_c.unfold().unfoldResult.hasOutputs$
$p_c.v()$	$= p_c.unfold().unfoldResult.v$
$p_c.n(l)$	= outcome of Algo. 4.6
$p_c.unfold()$	= outcome of Algo. 4.7

Table 4.11: Pseudo-state type. We use P_C, P_S and P_G to represent the pseudo-state types of resp. the Combinator, the **ioco**-Primer, and the *traces* Primer, and p_c, p_s and p_g to represent the corresponding pseudo-state instances. Furthermore, $l \in L^\delta$ is a label.

Algorithm 4.6: $P.n(l)$ for Combinator

input : P, a pseudo-state
l, a label
p_s, an **ioco** Primer that gives access to the Specification
p_g, a *traces* Primer that gives access to the observation objective
o, an IO-Oracle
output: P', the pseudo-state reached from P by l, unfolded

```
1 begin
2     ⟨n_s, v_s⟩ ⟵ p_s.next(P.m, l)
3     ⟨n_g, v_g⟩ ⟵ p_g.next(P.g, l)
4     if n_s = ⊥ ∧ n_g = ⊥ then
5         return ⟨⊥, ⊥⟩
6     P' ⟵ PS(n_s, n_g)
7     return ⟨P', P'.v()⟩
```

The Combinator pseudo-state type

In Table 4.11 we show the functionality offered by the pseudo-state type of the Combinator, and the definition of all its methods, except for n, which is given in Algorithm 4.6, and *unfold*, which is given in Algorithm 4.7. Below we first define the Combinator pseudo-state type, and then show and discuss n and *unfold*.

Definition 4.3.1
A pseudo-state of the Combinator is a tuple $\langle m, g, unfoldResult \rangle$, with

1. m, a pseudo-state returned by the **ioco** Primer;
2. g, a pseudo-state returned by the *traces* Primer;
3. *unfoldResult*, a tuple $\langle i, o, hasOutputs, v \rangle$ that holds information obtained from the **ioco** Primer and the *traces* Primer. The tuple contains
   1. i, the set of enabled input labels;
   2. o, the set of tuples of an enabled output label and a verdict;
   3. *hasOutputs*, a boolean that indicates whether one might choose to observe;
   4. v, the verdict associated with the current tester state.

After construction of a pseudo-state instance, before it is unfolded, its *unfoldResult* will have the value $\bot$, as shown in the definition of constructor PS in Table 4.11.

□

Method n Method n combines the results obtained by invoking the next function in both the **ioco** Primer and the *traces* primer. It either returns $\langle \bot, \bot \rangle$ —when in neither Primer a valid successor state is reached via label l—or it returns a new pseudo-state that contains the results returned by both Primers.

Method *unfold* Method *unfold* computes the enabled inputs and outputs (with the associated verdicts), and the verdict associated with the "current"

Algorithm 4.7: $P.unfold()$ for Combinator

input : P, a Combinator pseudo-state
p_s, an **ioco** Primer that gives access to the Specification
p_g, a *traces* Primer that gives access to the observation objective
o, an IO-Oracle
output: P, the pseudo-state, unfolded if it wasn't already unfolded

```
1  begin
2      if P.unfoldResult = ⊥ then
3          inputs ⟵ outputs ⟵ ∅
4          hasOutputs ⟵ false
5          if P.m = ⊥ then
6              if (P.g = ⊥) ∨ (ϵ ∉ p_g.out(P.g)) then
7                  verdict ⟵ ⟨fail, miss⟩
8              else
9                  verdict ⟵ ⟨fail, hit⟩
10         else
11             if P.g = ⊥ then
12                 verdict ⟵ ⟨pass, miss⟩
13             else
14                 verdict ⟵ ⊥
15                 inputs ⟵ p_s.in(P.m) ∩ p_g.in(P.g)
16                 o_s ⟵ {l | ⟨l, ⊥⟩ ∈ p_s.out(P.m)}
17                 o_g ⟵ {l | ⟨l, ⊥⟩ ∈ p_g.out(P.g)}
18                 foreach l ∈ (o_s ∪ o_g) do
19                     if l = ϵ ∧ l ∈ o_g then
20                         verdict ⟵ ⟨pass, hit⟩
21                     else if l ∉ o_s ∧ l ∈ o_g then        // is error'hit'?
22                         ⟨d, v⟩ ⟵ p_g.next(P.g, l)  // d ≠ ⊥ because l ∈ o_g
23                         if ϵ ∈ p_g.out(d) then
24                             outputs ⟵ outputs ∪ ⟨l, ⟨fail, hit⟩⟩
25                             hasOutputs ⟵ true
26                     else if l ∈ o_s ∧ l ∈ o_g then
27                         ⟨d, v⟩ ⟵ p_g.next(P.g, l)  // d ≠ ⊥ because l ∈ o_g
28                         if ϵ ∈ p_g.out(d) then
29                             outputs ⟵ outputs ∪ ⟨l, ⟨pass, hit⟩⟩
30                         else
31                             outputs ⟵ outputs ∪ ⟨l, ⊥⟩
32                         hasOutputs ⟵ true
33                     else if l ∈ o_s ∧ l ∉ o_g then
34                         outputs ⟵ outputs ∪ ⟨l, ⟨pass, miss⟩⟩
35         P.unfoldResult ⟵ ⟨inputs, outputs, hasOutputs, verdict⟩
36     return P
```

$$\mathsf{start}() = \mathsf{PS}(p_s.\mathsf{start}(), p_g.\mathsf{start}()).\mathsf{resultTuple}() \quad (4.28)$$
$$\mathsf{in}(P) = P.i() \quad (4.29)$$
$$\mathsf{hasOutputs}(P) = P.hasOutputs() \quad (4.30)$$
$$\mathsf{out}(P) = P.o() \quad (4.31)$$
$$\mathsf{next}(P, l) = P.n(l) \quad (4.32)$$
$$\mathsf{defNegVerdict}() = \langle \mathbf{fail}, \mathbf{miss} \rangle \quad (4.33)$$
$$\mathsf{defPosVerdict}() = \langle \mathbf{pass}, \mathbf{miss} \rangle \quad (4.34)$$

Table 4.12: Implementation of the DerivationEngine interface functions, in the Combinator; where p_s and p_g represent the Primer that give access to the specification resp. the observation objective.

pseudo-state. It does this using the in and out functions of both its Primer, and using the next function of the *traces* primer—it uses the latter in the computation of the verdict that is to be associated with an enabled output label: for that it needs to know whether "end-of-trace" is reached after that output.

While processing output transitions, it keeps track of whether there are outputs that do not lead to miss—this is used for the implementation of the hasOutputs interface function.

DerivationEngine interface implementation

In Table 4.12 we show how the Combinator implements the DerivationEngine interface functions, using the pseudo-state functions of Table 4.11. As in the case of random testing, the work is done by (methods of) the pseudo-state type.

Correctness

For the guided case we have essentially the same proof obligation as for the unguided (random testing) case (see page 118).

Optimization for infinite guidance

At the start of Section 3.5 we distinguished two kinds of guidance information: (1) guidance information that *directs* the test derivation, and (2) guidance information that *constrains* the test derivation. As mentioned there, the former can be treated as a finite set of finite traces, and the latter can be treated as a finite set of *infinite* traces.

We give a **hit** verdict when we reach the end of (a finite trace in) the guidance information, and, when computing the set of expected outputs, we associate a **hit** verdict with outputs, accordingly. However, to be able to associate **hit** verdicts with outputs, we have to look ahead in the guidance information: for each of the enabled outputs we invoke the next function of the *traces* Primer, and check whether, in the state thus reached, end of guidance is reached.

Reaching the end of guidance information, is, of course, only possible when the guidance information corresponds to a set of finite traces, i.e., only with guidance information that *directs* the test derivation.

For guidance information that is intended to *constrain* the test derivation, and that thus corresponds to a finite set of *infinite* traces, it is impossible to reach the end. Thus, for such guidance information, it is not useful to check whether we have reached the end. Therefore, when we know that guidance information only contains infinite traces, we use a version of Algorithm 4.7 that does not look ahead for end-of-guidance: it does not execute lines 22–25 and lines 27–30. Our tool implementation relies on the user of the tool to indicate (in the tool configuration) whether or not the guidance information contains only infinite traces.

4.4 DerivationEngine to access Off-Line Test Cases

We now discuss the DerivationEngine for test execution, i.e. the DerivationEngine that gives access to a test case.

Our design of this DerivationEngine aims to achieve the following:

1. support the "off-line testing" part of requirement 3 (design should be suitable for both on-line and off-line testing), by adapting our decomposition of Figure 4.4 to deal with test cases: extend Explorer and IO-Oracle components to be verdict-aware, such that a new *exec* primer can use them to provide access to a test case.

We assume that a test case is like we implicitly defined them in Chapter 3, i.e. as an LTS with the following features: (a) it is deterministic, and thus contains no internal (τ)-transitions; (b) it contains no unexpected outputs (i.e. outputs that lead to verdict **fail** or $\langle \mathbf{fail}, \mathbf{miss} \rangle$); and (c) verdicts are represented by self-loops with special verdict labels.

Note that such test case lacks one feature: it does not contain the default "positive" and "negative" verdicts. For the defNegVerdict and defPosVerdict DerivationEngine interface functions we must know, though, whether the "plain" default verdicts of Table 3.9 must be used, or the "guided" default verdicts of Table 3.10. From a test case, this information can only be inferred by inspecting

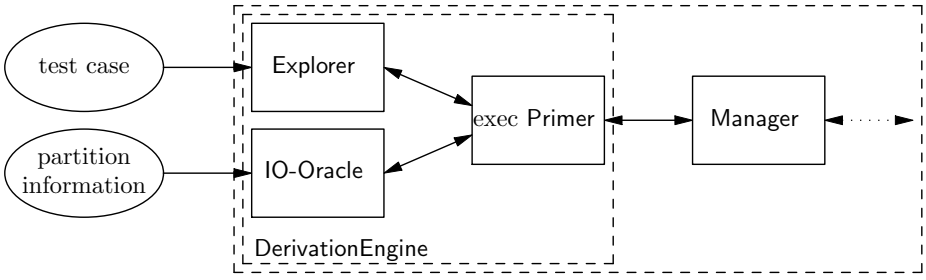


Figure 4.10: Decomposition of DerivationEngine into Explorer, Primer and IO-Oracle.

(at least one of) its verdict labels. For now we assume that the user knows whether "plain" or "guided" default labels must be used.

4.4.1 Components

Figure 4.10 shows the decomposition that we use. This is very similar to the one for random testing, shown in Fig. 4.4. The only difference is that, where in the case of random testing we had an **ioco** (or **uioco**) Primer, we now have an *exec* Primer that gives access to the test case.

The *exec* Primer is very similar to the *traces* Primer that we discussed in Section 4.3. The main difference is that the *exec* Primer also gives access to the verdicts in the test case.

4.4.2 Interfaces

Between (exec) Primer and Explorer: the Explorer interface

1. start $:\rightarrow S$
2. menu : $S \rightarrow \mathcal{P}((L_\tau \cup \{\delta\} \cup L_V) \times S)$

Table 4.13: Signature of Explorer interface functions, that can return a label that represents a verdict.

The return type of the Explorer interface is extended, to allow it to return verdict labels. This is the only change to the Explorer interface.

Between Primer and IO-Oracle: the IO-Oracle interface

There may be labels in a test case that represent verdicts (we represent verdicts that are associated with states using self-loops). As mentioned above, we extend the IO-Oracle interface with the ability to identify such labels. We assume that the "partitioning information" has been extended as well, and thus an initialised IO-Oracle instance has (in addition to functions isI, isU, isH, and isQ) a function that tests for verdicts: isV.

1. kind : $L^{\delta}_{\tau,V} \rightarrow \{\text{I}, \text{U}, \tau, \delta, \text{V}\}$

Table 4.14: Signature of IO-Oracle interface function that supports verdicts ($L^{\delta}_{\tau,V}$ represents $L_\tau \cup \{\delta\} \cup L_V$).

Ad 1: kind Function kind tells to what partition a given label belongs. It is defined as follows (with $l \in L_\tau \cup \{\delta\} \cup L_V$):

$$\mathsf{kind}(l) \;=_{\text{def}} \begin{cases} \text{I} & \text{if } isI(l) \\ \text{U} & \text{if } isU(l) \\ \tau & \text{if } isH(l) \\ \delta & \text{if } isQ(l) \\ \text{V} & \text{if } isV(l) \end{cases} \tag{4.35}$$

Provided by the Primer: the DerivationEngine interface

In the case of test execution, it is the Primer that realises the DerivationEngine interface, using the functionality offered by the Explorer and IO-Oracle.

4.4.3 Exec Primer algorithm

To realise the interface, we (again) refine the pseudo-state type, to obtain the one used by the *exec* Primer.

Signature:	
PS :	$S \to P$
m :	$P \to S$
resultTuple :	$P \to (P \uplus \bot) \times (L_V \uplus \bot)$
i :	$P \to \mathcal{P}(L_I)$
o :	$P \to \mathcal{P}(L_U^\delta \times (L_V \uplus \{\bot\}))$
v :	$P \to L_V \uplus \bot$
n :	$P \times L^\delta \to S \uplus \bot$
unfold :	$P \to P$
Definition:	
PS(S)	$= \langle S, \bot \rangle$ (a new pseudo-state)
P.m()	$= P.m$
P.*resultTuple*()	$= \langle P, P.v() \rangle$
$P.i()$	$= P.unfold().unfoldResult.i$
$P.o()$	$= P.unfold().unfoldResult.o$
$P.v()$	$= P.unfold().unfoldResult.v$
$P.n(l)$	$= P.unfold().unfoldResult.n_l[l]$
$P.unfold()$	$= P$ (unfolded, if not unfolded before, see Algo. 4.8)

Table 4.15: Pseudo-state type. P is a pseudo-state instance, and $l \in L^\delta \cup L_V$ is a label.

The *exec* Primer pseudo-state type

In Table 4.15 we show the functionality offered by the pseudo-state type of the *exec* Primer, and the definition of all its methods, except for *unfold* which is

Algorithm 4.8: *P.unfold*() for exec Primer

input : P, a pseudo-state
e, an Explorer that gives access to the test case
o, an IO-Oracle
output: P, the pseudo-state, unfolded if it wasn't already unfolded

```
1  begin
2      if P.unfoldResult = ⊥ then
3          verdict ⟵ ⊥
4          inputs ⟵ outputs ⟵ ∅
5          destmap ⟵ {l → ⊥ | l ∈ L^δ}
6          if P.m_d ≠ ⊥ then
7              foreach ⟨l, p'⟩ ∈ e.menu(P.m_d) do
8                  kind ⟵ o.kind(l)
9                  if kind = V then
10                     verdict ⟵ l
11                 else if kind = I then
12                     inputs ⟵ inputs ∪ {l}
13                     destmap[l] ⟵ p'
14                 else if kind = U ∨ kind = δ then
15                     n_v ⟵ ⊥
16                     n_o ⟵ e.menu(p')
17                     if ∃⟨l', p''⟩ ∈ n_o with o.kind(l') = V then
18                         n_v ⟵ l'
19                     outputs ⟵ outputs ∪ {⟨l, n_v⟩}
20                     destmap[l] ⟵ p'
21                 else if kind = τ then
22                     // should not happen
23         else
24             // should not happen
25         P.unfoldResult ⟵ ⟨inputs, outputs, destmap, verdict⟩
26     return P
```

given in Algorithm 4.8. We define the *exec* Primer pseudo-state type in Definition 4.4.1.

Definition 4.4.1
A pseudo-state of the *exec* Primer is a tuple $\langle m_d, unfoldResult \rangle$, with

1. m_d, an LTS state: a state directly reachable, like the initial state, or the destination of an observable transition;
2. *unfoldResult*, a tuple $\langle i, o, n_l, v \rangle$ that holds the result of "unfolding" (explained below) the state in m_d. The tuple contains
   1. i, the set of enabled input labels;
   2. o, the set of tuples of an enabled output label and a verdict;

3. $n_l : L^\delta \to S \uplus \{\bot\}$, a mapping from labels to the state that is directly reachable via the label (a single state, because a test case is deterministic);
4. v, the verdict associated with the current tester state.

After construction of a pseudo-state instance, before it is unfolded, its *unfoldResult* will have the value $\bot$, as shown in the definition of constructor PS in Table 4.15.

□

Method *unfold* Like the *unfold* method of the Combinator, the *unfold* method of the *exec* Primer computes the enabled inputs and outputs (with the associated verdicts), and the verdict associated with the "current" pseudo-state.

Like the *unfold* method of the Combinator, it looks ahead to obtain verdicts that are to be associated with outputs.

Algorithm 4.8 relies on a test-case being deterministic, and thus 1. it does not handle internal (τ) transitions; 2. it assumes that, from any given state, with any label, at most one state in the test case is reached; 3. it assumes that there is at most one verdict associated with any state in the test case. All three are valid assumptions, given how we derive test cases.

DerivationEngine interface implementation

In Table 4.16 we show how the Primer implements the DerivationEngine interface functions, using the pseudo-state functions of Table 4.15.

One thing to point out is how we define the hasOutputs function. Each test step in the test case will either try to apply a stimulus, or obtain and check an observation. The Manager should just honour whatever choice is in the test case. The Manager uses hasOutputs when it decides between stimulating and observing. Thus, hasOutputs should only return true when the test step is an observation. Hence the additional check whether the test step has any inputs in Equation 4.38.

$$\mathsf{start}() = \mathsf{PS}(e.\mathsf{start}()).\mathsf{resultTuple}() \tag{4.36}$$

$$\mathsf{in}(P) = P.i() \tag{4.37}$$

$$\mathsf{hasOutputs}(P) = P.i() = \emptyset \wedge P.o() \neq \emptyset \tag{4.38}$$

$$\mathsf{out}(P) = P.o() \tag{4.39}$$

$$\mathsf{next}(P, l) = \begin{cases} \langle \bot, \bot \rangle & \text{if } P.n(l) = \bot \\ \mathsf{PS}(P.n(l)).\mathsf{resultTuple}() & \text{otherwise} \end{cases} \tag{4.40}$$

$$\mathsf{defNegVerdict}() = \mathbf{fail} \text{ or } \langle \mathbf{fail}, \mathbf{miss} \rangle \tag{4.41}$$

$$\mathsf{defPosVerdict}() = \mathbf{pass} \text{ or } \langle \mathbf{pass}, \mathbf{miss} \rangle \tag{4.42}$$

Table 4.16: Implementation of the DerivationEngine interface functions, in the *exec* Primer; where e represents the Explorer that give access to the test case.

Correctness

For the case of test execution we have essentially the same proof obligation as for the unguided (random testing) case (see page 118).

4.5 Summary

In this chapter we showed how we do test derivation, how we deal with τ-cycles, and how we support visualisation, and we discussed our support for a number of modelling languages.

Chapter 5

Test Execution Engine

In Chapter 3 we gave an overview of the functionality and architecture of TorX, and we presented the interface that is offered by the Adapter. In this chapter we discuss how this interface can be provided.

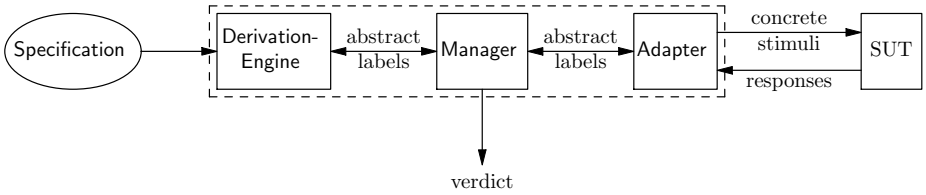


Figure 5.1: Position of the Adapter in the high-level architecture. Interaction between Manager and Adapter is using the Adapter interface functions applyStim and getObs, using labels from the model to represent interactions between Adapter and SUT. Interaction between Adapter and SUT uses whatever means of interaction the SUT offers.

The rôle of the Adapter is to interact with the system under test while a test case is being executed, where the test case is represented using labels of the model, as depicted in Figure 5.1. Two main functions of the Adapter are thus:

- interacting with the system under test, represented by *concrete stimuli* and *responses* in Fig. 5.1, and
- mapping between these interactions and their representation as (*abstract*) *labels* of the model.

From these two functions, we see that an Adapter instance is not only IUT-specific, but also model-specific, in particular, specific to the model labels. In addition, an Adapter instance may also be test-architecture-specific. Recall that a test architecture (see on page 8) specifies via which interfaces the test tool is supposed to interact with the IUT, and the concepts "test context", PCO (point of control and observation), IAP (implementation access point), and

SUT (system under test). Thus, a test architecture dictates which PCOs an Adapter needs to have, to be able to execute test cases in accordance with the test architecture. Therefore, when referring to Adapter functionality in general, we will refer to *an* Adapter rather than to *the* Adapter. We now look at the constraints that we impose on the design of an Adapter.

Design constraints The design of an Adapter is constrained, on the one hand, by the interface that it has to provide, as given in Table 3.12. On the other hand, it is constrained by requirements that we discussed in Chapter 1, in particular:

1 the tool should be based on **ioco** theory;
5 the tool design should be independent from particular modelling languages;
8 the tool design should make no assumptions about the SUT, except that it is a reactive system;
13 it should be easy to create a simple model (like an automaton) for use with the tool;
15 it should be possible to use a simulated model as system under test;
16 it should be simple to connect the tool to toy implementations;
19 the design should allow use of modelling languages with suitable expressive power;
21 the tool should produce/keep test execution data for analysis;
24 it should be easy to connect the tool to the system under test (subsumes 16).

Requirement 1 directly affects (the design of) an Adapter: it requires the Adapter to be able to observe quiescence. Requirements 5, 13 and 19 illustrate that, in general, we support multiple modeling languages. Thus, for a single SUT, we may have several models, each with a different label set. In that case we need, at least conceptually, multiple Adapter instances, that differ only in the label-to-interaction mapping. Thus, our Adapter design must support creation of such family of Adapter instances. Requirement 8 states that we do not limit ourselves to specific classes of systems, as long as they are reactive. This means (a) that we cannot make any assumption on how the SUT expects to interact with its environment (whether using a network protocol, or an API, or via some sort of physical interaction, or anything else); and (b) that the SUT will take the initiative to produce output as it sees fit (we only know that once it is quiescent, it will stay quiescent until further input is given to it). To interact with a SUT, an Adapter must conform itself to the interfaces via which the SUT interacts with its environment—this includes establishing a means of communication with the SUT, as we discuss below (see "Common Adapter functionality"). In addition to requirement 8 (no assumptions about SUT), we also have requirement 15 and requirement 16 that demand specific support for resp. "simulated models" and toy implementations—we discuss both below; there we also discuss how we interpret "toy implementation". To support requirement 21 we include the concrete interactions between Adapter and SUT in the test run log, with time stamps.

Adapter support We deal with these requirements

1. by having specific built-in Adapter instances for the required two cases:
   - use of simulated model as IUT;
   - use of "toy implementation" as IUT.
2. by offering the torx-adapter interface [tora] for using external Adapter programs (in spirit this is similar to the torx-explorer interface that allows the use of external Explorer programs), and
3. by giving a general design for an Adapter. We introduce the design by discussing three Adapter instances; in the design we generalise the functionality encountered in these examples. Of course, the design is also inspired by Adapter instances used in other case studies that we have worked on, see Chapter 8.

Common Adapter functionality

Before we discuss the Adapter examples, and our general design, we look at functionalities that all Adapter instances have in common. To do its job, each Adapter must be able to provide the two main Adapter functions:

1. interact with the IUT (this includes establishing a means to interact with it), if necessary through a test context (i.e., SUT $\neq$ IUT);
2. map between model labels and interactions with the SUT.

In addition, it must also be able to

3. observe quiescence;
4. provide the Adapter interface of Table 3.12;
5. synchronise: the SUT takes the initiative for outputs, and the Manager takes the initiative for all operations on the Adapter interface.

We use this list to structure the discussion of each of the Adapter examples, and as basis for our Adapter design. We discuss establishing a means to interact, observation of quiescence, and synchronisation below.

Establishing a means to interact Typically, we let the Adapter take the initiative to establish a means to interact with the IUT at the start of a test run, i.e. in the start interface function implemented by the Adapter. For an IUT that interacts on standard input and output, the easiest (only?) way to establish communication is by letting the Adapter start the IUT and create pipes to its standard input and output. Starting the IUT at the start of the test run has the benefit that, at the start of the test run, the IUT is 'automagically' in its initial state. Also for an IUT that interacts (also?) in another way, e.g. via one or more network connections, the start interface function is the place to establish such network connection(s), whether actively (by connecting to a network interface provided by the IUT), or passively (by establishing a network interface to which the IUT may connect). In such case, the IUT may already be active at (before) the start of the test run, and thus, at the moment that a means of interaction is established, the IUT need not be in its initial state. We assume that in such case (a) there is a way to reset the IUT to a known state, and the model used for testing starts with the abstract representation of this reset, or (b) there is

a way for the Adapter to obtain (information about) the state of the IUT (this we used in the Easylink case study, discussed in Appendix B.2).

We also assume that the stop interface function undoes the work done by start, i.e. it will stop an IUT that it started, or disconnect any connection that it made.

Observing quiescence Typically, we observe quiescence by starting a timer, and waiting to see whether the IUT produces output before the timer expires.

On the one hand, from the viewpoint of theory, all that matters is that we wait long enough. If we wait too short, we might make the mistake of concluding that an IUT is quiescent, while it is only slow. If, at that moment, quiescence is not expected, we get an immediate **fail**. If, instead, quiescence is expected at that moment, e.g. because the model is non-deterministic and expects both output and quiescence, we do not get an immediate **fail**. However, if then, after the observation of quiescence, the "slow" output arrives, we still get a **fail** verdict, because once quiescence has been observed, output is not allowed. Only when between the (mistaken) observation of quiescence and the observation of the slow output, already a stimulus has been given, then we may not get a **fail** verdict, because then to the tester it may appear that, after the (mistaken) observation of quiescence, the stimulus caused (triggered) the slow output. Thus, if the quiescence timeout value is too short, we typically reject a correct (but slower than expected) IUT, i.e. then our testing has become unsound.

On the other hand, from the viewpoint of test efficiency, we should wait as short as possible. As soon as we wait for any significant amount of time to observe quiescence, typically—for any IUT that occasionally is quiescent—overall wall-clock testing time is, to a large extent, determined by (spent) waiting for quiescence.

In Section 7.5.6 we look at the test execution times of a case study; the effect of waiting for quiescence on the time between test steps is clearly visible there.

Synchronisation – races The Adapter has to deal with the fact that the SUT takes the initiative for outputs, and the Manager takes the initiative for all operations on the Adapter interface. Thus, output obtained from the SUT has to be stored until the Manager asks for it. Moreover, when communication between SUT and Adapter is asynchronous, we have to consider the possibility of races: messages, that are approximately sent concurrently may overtake each other. So, it may be hard, or impossible, to establish in the Adapter in what precise order the SUT interacted with its environment, i.e., with the Adapter. And, as we see in the examples in Section 5.1, the presence of a test context that behaves as one or more FIFO buffers does not make things easier. We distinguish two kind of races:

1. input-output races, where both SUT and Adapter try to communicate, by e.g. sending a message, at approximately the same moment, and
2. concurrent output races, where the SUT provides multiple outputs at approximately the same moment.

Typically, we deal with such races outside the Adapter. To be able to cope with, or prevent, a race in the Adapter, we would first have to be able to detect that it has happened (or is about to happen), and this is often not possible. We have experimented with time-stamping interactions in the Adapter, and using these timestamps, together with a small queue of recent interactions in the Manager, to adjust apparent reordered interactions, but we do not discuss this further. In practice, we typically deal with races at the model level: by avoiding input-output non-determinism in the model, we can try to avoid input-output races, and by including FIFO-queues in the model, we can make it—and the test cases derived from it—robust against message reordering, both for input-output races and concurrent output races. In our experience, this is not only the easiest solution, but also the most complete one: it also works for reordering caused by a test context—unless the IUT or the test context timestamp interactions with the IUT, we do not see how an Adapter may solve such reordering.

Remainder of this chapter First, in Section 5.1, we discuss three Adapter instances, to illustrate the functionality that an Adapter has to offer, and to illustrate additional requirements for the Adapter design. Then, in Section 5.2, we generalise the functionality, seen in the examples, to our general Adapter design.

5.1 Adapter Examples

To illustrate the functionality that an Adapter has to offer and to elicitate additional requirements, we discuss three Adapter instances, as shown in Table 5.1:

1. for "toy implementations", in Section 5.1.1,
2. for a conference protocol entity, in Section 5.1.2, and
3. for a software bus server, in Section 5.1.3.

used for:	Toy Impl.	Chatbox	Software Bus
#IAP:	1	2	1
#PCO:	1	3	3
interaction via:	stdin/stdout	stdin/stdout + UDP	TCP
interaction set-up:	static	static	dynamic
mapping:	-	data + address	address
mapping configuration:	-	static + dynamic	dynamic
possible races:			
in-out non-determ.	x	x	
concur. msg reord.		x	x
discussed in:	Sect 5.1.1	Sect 5.1.2	Sect 5.1.3

Table 5.1: Overview of the examples that we discuss.

Ad 1: "toy implementations" The first Adapter communicates model labels over the standard input and output of an IUT. For this Adapter, the map-

ping between interactions and labels is trivial, and, regarding potential races there is only potential input-output non-determinism.

Ad 2: conference protocol entity The second Adapter communicates with the IUT in two ways: over its standard input and output, and over UDP [Pos80]. In both cases, encoded messages are exchanged. The main difference with the first Adapter is that, where for communication over standard input and output it suffices to just write and read messages, with communication over UDP also UDP addresses (source and destination) play a role. Moreover, due to the use of encoded messages, the mapping between interactions and labels is less trivial than with the first Adapter. Finally, because both Adapter and IUT send and receive messages at multiple PCOs resp. IAPs, there is the possibility of reordering of messages that are sent "at approximately the same moment" over different communication interfaces.

Ad 3: software bus server The third Adapter communicates with the IUT over TCP [Pos81], using model labels. The main difference with the other two Adapter instances is that this Adapter actively opens and closes TCP connections during a test run, i.e. the "interaction infrastructure" changes dynamically during a test run. Thus, this Adapter is stateful: it maintains interaction infrastructure state.

Below we discuss the three example Adapter instances in more detail, where we use the same structure in each case: first we introduce the SUT, and then we discuss the Adapter, where we, in turn, discuss each of the five common Adapter functionalities that we gave above in the introduction.

5.1.1 Stdin/out Adapter for Toy Implementations

Here we discuss the Adapter that aims to fulfil Requirement 16 (connect to toy implementations easily). As we will see, this Adapter is made for IUTs that communicate using model labels over standard-input and output. We first discuss the IUT, and then the Adapter.

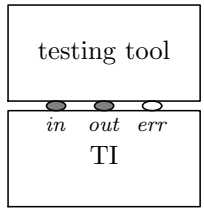


Figure 5.2: Test architecture for toy implementation (TI). The testing tool interacts with the TI at standard input and standard output, and consumes the diagnostic output of the TI at its standard error.

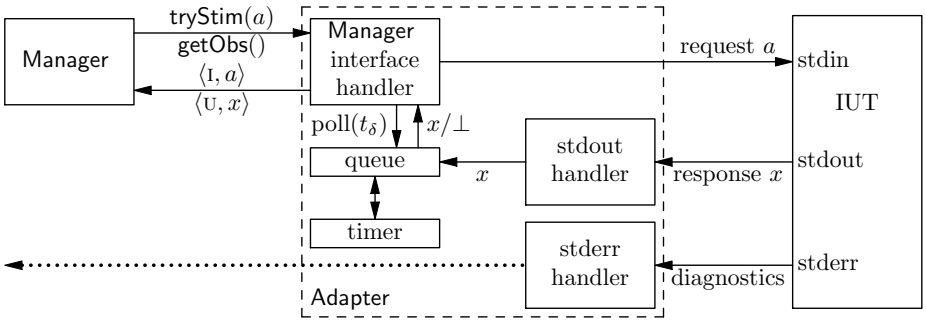


Figure 5.3: Adapter for toy implementations.

The IUT: a Toy Implementation

Fulfilment of the requirement to "connect to toy implementations easily" starts with defining what a toy implementation is. To us, a toy implementation is an implementation, that is developed to exhibit a certain behaviour, i.e. to perform certain series of interactions with its environment. Typically, such intended behaviour will be described in a design; part of such design can be an (LTS) model that describes the intended behaviour in terms of input- and output labels.

Given the focus on exhibiting a certain behaviour, the way in which such implementation interacts with its environment should be as simple as possible. Given that the behaviour is defined in terms of input- and output labels, it feels natural to let the interaction take place using input- and output labels. The remaining question then is: how does the interaction take place? We want to make as few assumptions on the implementation as possible, to give maximum implementation freedom. Therefore, we let the interaction take place over the standard input and standard output of the implementation, using lines of text, one label per line[1]. The implementation can then use standard error to write diagnostics etc. With this choice, a user can interact with the implementation by just starting it, typing at it, and reading its responses. Also, there is ample implementation freedom: communication over standard input- and output can easily be implemented in any programming language.

Testing a Toy Implementation

In this case, the SUT coincides with the IUT; there is no test context. The test architecture is depicted in Figure 5.2.

[1] An (obvious) alternative would be to communicate model labels over TCP. TorX supports this with two additional Adapter instances: one for the case where the IUT opens a listening socket and waits for its environment to connect (this Adapter is a TCP client), and one for the case where the IUT expects the environment to open a listening socket, to which it then connects (this Adapter is a small TCP server).

The Adapter

We now discuss our Adapter for this kind of toy implementations, see Figure 5.3.

Ad 1: Interact with IUT This is the first of the two main Adapter functions that we mentioned in the introduction to this chapter. We let the Adapter "play the role of the user", i.e. we let it provide the environment that the IUT expects. Thus, it has to be able to start the IUT program, write labels as lines of text to its standard input, and read labels as lines of text from its standard output. This Adapter does all this as follows.

When the Adapter is started, it starts the IUT program in such a way that it can write to the IUT program standard input, and can read from the IUT program standard output and standard error. The Adapter runs three threads:

- the main thread, on which it interacts with the Manager, and on which it applies stimuli to the IUT (because this Adapter is just a module in the test tool, the Adapter main thread coincides with the test tool main thread);
- a thread to read the standard output of the IUT program;
- a thread to read the standard error of the IUT program.

The two reader threads can wait for output from the IUT, independently, while at the same time stimuli are applied.

Synchronisation between the threads happens via a shared FIFO queue: the stdout-handler appends label strings to it, and the Manager interface handler gets them from it. The shared queue has additional synchronisation functionality: in the poll request to get the first item of the queue, a time-out value can be provided. When the queue is empty, and the timeout value is greater than zero, the poll request will wait until either an item is added to the queue, or the timeout time has passed, whatever happens first. It will return the item just added in the former case, and $\perp$ in the latter.

The stderr-handler just writes everything that it reads from the IUT to the standard error of the tester tool. The reason to nevertheless intercept the IUT standard error are two-fold. Firstly, we may use it to provide additional convenience to the user, by e.g. filtering out specific diagnostics and showing them to the user. Secondly, it avoids blocking: when this output is not consumed, , the operating system may suspend the execution of the IUT program when the IUT program writes to standard error and the corresponding write buffer is full.

Ad 2: Mapping between labels and interactions This is the second main Adapter function. In this Adapter, the mapping is trivial.

To encode a stimulus, it suffices to add a new-line character to the end of the string representation of the label that is the argument of the tryStim interface function. To decode an observation, it suffices to remove the new-line character from the line of text which is read from the standard output of the IUT program.

In this case, the encoding can be done by the Manager interface handler, just before it applies a stimulus, by writing the encoding result to the IUT standard input. The decoding can be done by the stdout handler, just before it adds the decoding result to the queue.

Ad 3: Observing quiescence This can be seen as a special case of the second main Adapter function: it has to map *lack of interaction* onto a label. In this Adapter, this is done as follows.

This Adapter uses the poll functionality offered by the shared queue to observe quiescence. The Adapter is configured with a quiescence time-out value t_δ. When the Manager requests an observation, the Manager interface handler requests an observation from the queue using poll, with a time-out value t_δ. When the queue returns an observation, it is returned to the Manager; when the queue returns $\perp$ (time-out), an observation of δ is returned to the Manager.

Ad 4: Providing the Adapter interface The Manager interface handler implements the Adapter interface functions. As discussed above (ad 3) and below (ad 5) it uses the queue to obtain observations (when handling getObs), and to check for pending ones (when handling tryStim). Both getObs and tryStim only return once they have an observation, or quiescence, to return, or once the stimulus has been applied.

Ad 5: Synchronisation When the Adapter is handling a tryStim request to try to apply a given stimulus, before applying the stimulus, it first checks whether the queue contains a pending observation (using poll with a time-out value of 0). If there is a pending observation, the Adapter returns it, and ignores the given stimulus. Otherwise, it applies the given stimulus by writing it to the standard input of the IUT program. Once the Adapter has written the stimulus, the stimulus is out of its control, and it reports that it has applied the stimulus.

Potential improvements When requesting observations from the queue, this Adapter always uses the same t_δ value for the queue poll request.

The theory does not say anything about the quiescence timeout value; for practical reasons—avoiding incorrect test results, as we discussed in the introduction to this chapter—we only have to be careful to not choose a timeout value that is too short.

Thus, to speed up testing, by reducing the time that we spend, waiting for the quiescence timer to expire, the Adapter could remember at what time the latest interaction with the IUT took place, and, for the poll request timeout value, subtract the time that passed since last interaction from t_δ. That approach we also used in other Adapter instances.

5.1.2 UDP Adapter for a Conference Protocol Entity

The *conference protocol* is a simple chat box protocol that was designed for a course on protocol implementation. We created and tested a number of conference protocol entity (CPE) implementations, as discussed in [BFdV+99, DRS+00]. A detailed description of the conference protocol, and our implementations of it, can be found at [Feea, Feeb].

The main differences between (the Adapter for) a Toy Implementation and (one for) a CPE are (1) how they interact, and (2) what they communicate. Whereas a Toy Implementation interacts only via standard input and output,

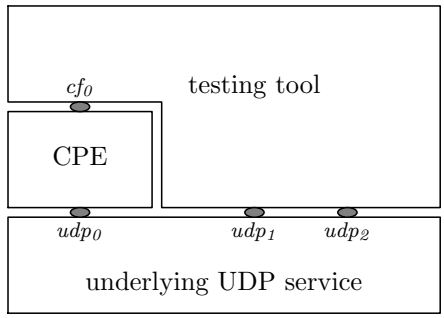


Figure 5.4: Test architecture for conference protocol CPE. The CPE has two IAPs: cf_0 and udp_0. The testing tool interacts directly with the CPE at PCO (IAP) cf_0 and via the UDP service at PCOs udp_1 and udp_2. The UDP service forms the test context.

a CPE communicates both via standard input and output, and via UDP. And, where a Toy Implementation communicates using model labels, a CPE communicates via messages with a specific encoding. As a consequence, the CPE Adapter contains, in addition to functionality to interact on a CPE's standard input and output, also functionality to interact via UDP, functionality to map between encoded messages and model labels, and functionality to pass encoded messages to the right interaction handling (sub-)component.

We now first give an overview of the functionality of the CPE, then we describe how we tested CPE instances, after which we describe the Adapter.

The IUT: a Conference Protocol Entity

Conference protocol service The conference protocol allows conference users to join a named conference, exchange messages with conference partners, and leave the conference they are in. It uses UDP[Pos80] as underlying service. A user can participate in at most one conference at a time. When a CPE instance is started, it is given the set of potential conference partners, as a set of UDP addresses (internet host name or IP address, together with a UDP port number). This set is then "frozen" during the CPE run.

Interfaces The CPE has two interfaces with its environment, indicated as cf_0 and udp_0 in Figure 5.4. At interface cf_0 it interacts with its user, and at udp_0 it interacts with the underlying UDP service. The CPEs use the underlying UDP service to communicate among each other to provide the conference service.

Messages At cf_0 four types of user messages are exchanged (join, leave, datareq and dataind), and at udp_0 four types of UDP messages are exchanged (join, leave, answer, data). These messages contain as parameters *user names*, *conference names*, and *user message data*, where the user and conference names consist of a 10-character string, and data messages of up to 256 octets. For details about these messages, and their encoding, we refer to [Feea, Feeb]; the

format of the user messages exchanged at cf_0 is very similar to the UDP messages.

Testing the Conference Protocol Entities

For all our CPE testing we used the same architecture. We tested with models in multiple modelling languages; here we mention LOTOS and Promela. Because the labels of the LOTOS model differ from those of the Promela model, we needed (at least conceptually) distinct—though very similar—per-modelling-language Adapter instances.

IUT instances We tested a "family" of 28 CPE implementations. The family consists of one assumed-to-be-correct implementation, and 27 mutants that have been derived from the correct one by introducing (known) errors. All CPE implementations can be tested with the same Adapter instance.

Test architecture The test architecture is depicted in Figure 5.4. We let the testing tool play the role of the user (interacting at cf_0), and of two peer CPEs (interacting at udp_1 and udp_2). The testing tool interacts directly with the CPE at cf_0 (the CPE's standard input and output) and via the underlying UDP service at udp_1 and udp_2. We chose to test via the underlying UDP service, because that was the easiest way to interact at interface udp_0. (To directly interact with the CPE at that interface, it would have been necessary to either change the CPE to directly provide access, or to replace the UDP networking library that the CPE uses by one that provides direct access. Neither of these options looked particularly inviting.) In principle, the UDP service is unreliable, but we assumed that, when using it over the loopback interface (i.e. the UDP packets are only moved around within the same machine, without actually travelling over the network), it will perform in a reliable way—just as we always assume that the test context behaves correctly. Our experimental results did not give us reason to regret this assumption—otherwise, we could have gotten inappropriate **fail** verdicts, caused, not by errors in an IUT, but just by unreliability of the UDP connection; i.e., we could have gotten unsound test results.

Models We tested the CPE implementations using models in LOTOS and Promela. The test architecture is reflected in the model. The models describe a CPE as a service that continuously processes messages, one message at a time. For each message that it processes (received from user or from underlying UDP service), it updates its internal state, and sends out any corresponding responses (to the user, and/or to the underlying UDP service).

In the LOTOS model we modelled also the test context, as FIFO queues, to represent the message reordering that the queue context may induce. For example, two messages, sent from udp_0 to udp_1 and udp_2 at the same moment (in response to the same stimulus), may arrive in any order. In the Promela model we explicitly modelled that messages sent by the IUT together (at the same time, in response to the same stimulus) may be observed in any order.

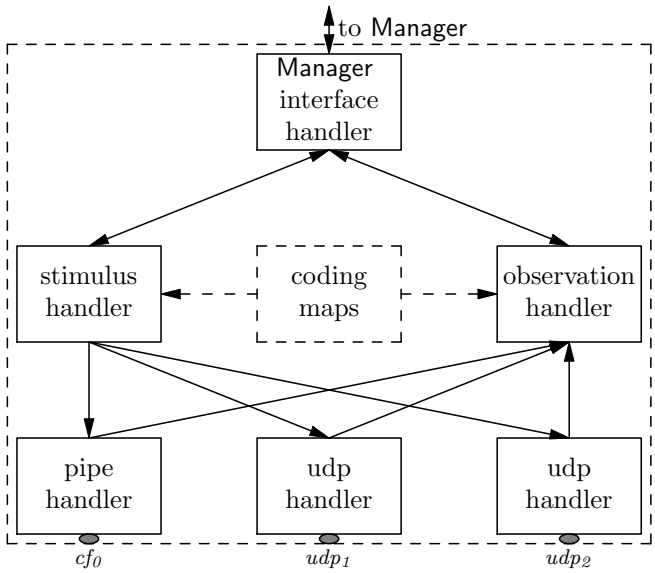


Figure 5.5: Overview of conference protocol adapter

In the model we made one simplification: for all messages, for each message parameter, we used an enumeration with only a few (three or four) different values. We made this simplification for two reasons. Firstly, because we considered it (practically) *unfeasible* to enumerate all possible data value combinations. For example, without this simplification, when a join user message would be enabled, the list of enabled stimuli would contain a label for each possible combination of a 10-character user name and a 10-character conference name. Secondly, because we considered it *unnecessary* to enumerate all possible data value combinations, because our focus was on testing the control paths of the CPE, for which a limited number of distinct data values suffices (for example, to check whether the CPE correctly maintains its set of conference partners).

Below we describe how we map, in the Adapter, between these enumeration values and concrete message parameter values. Choosing the concrete message parameter values amounts to test data value selection.

The Adapter instances

Also the two CPE Adapter instances (for LOTOS resp. Promela) must perform the 5 functionalities that we mentioned in the discussion of the Toy Implementation Adapter. Both Adapter instances have the same structure; they only have small differences in the encoding and decoding (mapping) components. In the description below, we first describe the overall structure of the Adapter, and then discuss how stimuli are applied and responses are handled.

Adapter structure The main structure of the Adapter is shown in Figure 5.5. It consists of a handler for the requests of the Manager, a stimulus handler, an

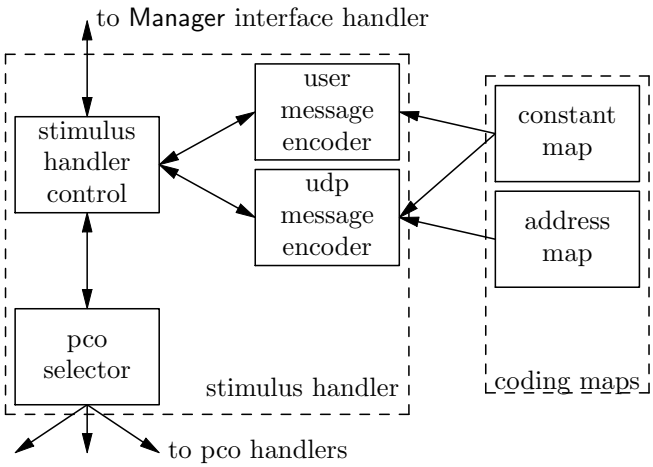


Figure 5.6: Stimulus handler of conference protocol adapter, with coding maps.

observation handler, and, for each PCO, a PCO-type specific interaction handler (i.e. for UDP and for pipes).

Ad 1: Interact with the IUT Each UDP handler binds a socket and lets the operating system choose a UDP port number for it. This gives it its UDP address (ip address and UDP port number). Each UDP handler concurrently waits for stimulus application requests from the stimulus handler, and for incoming datagrams on its socket. When a stimulus has to be applied, the UDP handler is given an octet string, together with the UDP address to send it to. As soon as a datagram arrives, it forwards the received octet-string, together with the UDP address of the sender, to the observation handler.

The pipe handler starts the CPE program and connects pipes to the standard input and standard output of the program. Before starting the CPE, it updates the potential conference partner configuration file for the CPE with the UDP addresses of the UDP interaction handlers. It concurrently waits for stimulus application requests from the stimulus handler, and for data that appears on the pipe connected to the standard output of the CPE. When a stimulus has to be applied, the pipe handler is given an octet-string to write on the pipe that is connected to the standard input of the CPE. As soon as data from the CPE is obtained, it is forwarded to the observation handler.

Ad 2: Mapping between labels and interactions We separately discuss this for stimuli and for observations.

2A: Stimulus handler When a stimulus has to be applied, the stimulus handler (see Fig. 5.6) is given a model label to apply.

First it obtains a concrete representation of the abstract stimulus that it can pass to the interaction handler, to pass it to the CPE. It obtains this representation from one of its two encoders. One encoder is used for the messages

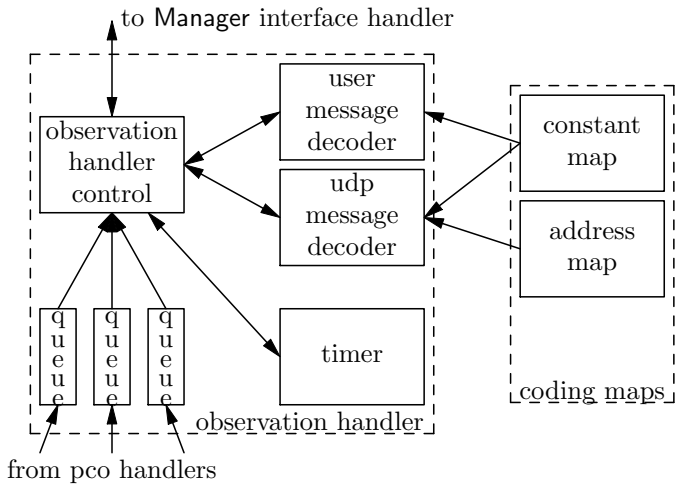


Figure 5.7: Observation handler of conference protocol adapter, with coding maps.

that are passed at cf_0, the other for the UDP messages that are passed at udp_1 and udp_2. The encoder for UDP messages returns a tuple: the message to be sent, together with the remote UDP address. The stimulus handler chooses an encoder based on the structure of the abstract action (label from the model). The abstract actions are chosen such that their prefix (the *gate* in LOTOS terminology, or *channel* in Promela) indicates the interaction mechanism that is to be used.

Both encoders consult a translation map that maps between abstract label constants for user names and conference names and concrete message parameters, as mentioned above. Moreover, the encoder for UDP messages consults a translation map that maps between the UDP addresses that are used during the test run and the abstract representations that are used in the model. Part of this map is dynamically constructed for each different test run, because the UDP addresses for udp_1 and udp_2 are chosen by operating system at the start of a test run, i.e. the Adapter is stateful.

The stimulus handler looks at the label to determine the PCO at which the stimulus has to be applied. For the stimuli that are to be passed over cf_0 there is only one possibility; for the stimuli that are to be sent over UDP it looks at the abstract representations of the UDP addresses in the labels.

Finally, it passes the encoded stimulus to the interaction handler for the PCO at which the stimulus has to be applied.

2B: Observation handler The observation handler (see Fig. 5.7) has a queue for each of the interaction handlers in which pending observations are kept until

they are processed[2]. When a request for an observation is received the first observation of one of the queues that has pending observations is decoded and returned. If no observation is pending a timer is set. When no observation has been obtained when the timer expires a quiescence observation is synthesised.

The decoder associates a decoding function with each PCO. The decoding functions consult the same maps for data constants and addresses as the encoding functions do.

Keeping separate queues for pending observations creates the possibility that they get reordered. In this case that was not as much an issue as it may seem on first sight, because similar reordering of observations may also take place due to the underlying UDP service and the pipe between CPE and Adapter, and therefore the models that were used for test derivation took the possibility of reordering into account, as mentioned above in the discussion of the models.

Ad 3: Observing quiescence As mentioned above, quiescence is observed by the observation handler, when the observation queues are empty, and the timer expires before any item is added to a queue.

Ad 4: Providing the Adapter interface The Manager interface handler implements the Adapter interface functions.

Ad 5: Synchronisation When handling tryStim, the Manager interface handler asks the observation handler for an observation, and instructs it to only return a pending one, without setting the timer, i.e. to not wait for a "fresh" observation when there is not one pending. If an observation is returned, the Manager interface handler returns it to the Manager (and discards the stimulus); otherwise it applies the stimulus, and acknowledges this to the Manager.

Adapter variants due to multiple modelling formalisms We used models in two different modelling languages in the case study, LOTOS and Promela. The models in the two languages were made by different persons, such that each model had its own syntactical representation for a particular interaction with the system under test. This led to separate sets of coding functions, to compensate for the different structure of the labels, and to separate mapping tables, to compensate for the different model representations for the interaction data value constants.

Bibliographical note The original Adapter instances for the conference protocol predate the change that made test cases input-enabled. The Adapter description here extends on these Adapter instances, to allow input-enabled test cases.

[2] The separate queues were not there by design, but as an artefact of the chosen implementation for the connection between the interaction handlers and the other components of the Adapter. We prefer a single queue, for how it orders observations.

5.1.3 TCP Adapter for a Software Bus Server

We now discuss the Adapter for testing a software bus server (SBS). In Chapter 7 we discuss the design, implementation, and testing of the SBS in detail. As we discuss there, we actually created and tested two SBS implementations, i_1 and i_2, each using a dedicated Adapter a_1 resp. a_2. Here we limit the discussion to implementation i_2 and Adapter a_2; here we will typically refer to them as "the IUT" resp. "the Adapter".

Below we first discuss the IUT, then we discuss how we tested it, after which we discuss the Adapter.

The IUT: a Software Bus Server

The software bus server (SBS) allows clients to communicate with each other without having to create connections between each pair of clients that wants to communicate: each client only has a single connection, to the server. The software bus uses TCP [Pos81] as underlying service.

Like the conference protocol entity (CPE) that we discussed in the previous section, the SBS is tested via the underlying service. However, where for the CPE all "interaction infrastructure" (connection to standard input and output of the IUT, creation of UDP listening sockets) was set up at the start of a test run, and did not change during the test run, for the SBS the interaction infrastructure is dynamically updated during a test run. With the SBS, a client may connect, exchange messages, and then disconnect. And then connect again, etc.

This difference also shows in the respective models: whereas the model for the CPE only contains actions (LTS labels) that represent message exchange between CPE and its environment, the model for the SBS also contains actions that represent connect and disconnect.

Other functionality of the SBS includes a notification service, and a service-announcement service. The notification service allows a client to ask the server to tell it (by sending it a message) about given activity of other clients, like connecting and disconnecting. The service-announcement service allows a client to tell the server what services it (the client) can perform. The server forwards such announcement to other connected clients. In the messages exchanged by server and clients, each connected client is identified by an ID (identification number). A client gets such ID from the server, when it has connected. In Chapter 7 we discuss the software bus server, and the services that it provides, in more detail.

Testing the Software Bus Server

Implementation i_2 implements all SBS behaviour. However, because with i_2 we were only interested in testing the interaction behaviour, and estimating testing quality, by measuring test coverage, we did not implement the message encoding and decoding, but let i_2 communicate using model labels.

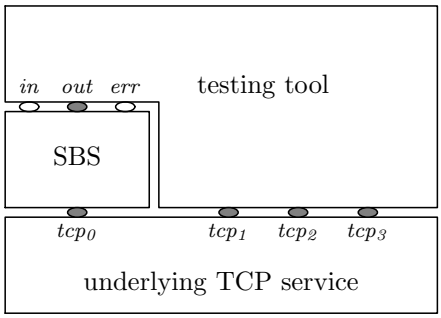


Figure 5.8: Test architecture for software bus server (SBS). The testing tool interacts with the SBS through the underlying TCP service at PCOs tcp_1, tcp_2 and tcp_3. The testing tool observes the standard output of the SBS at *out*, because there the SBS announces its TCP address (i.e. the address of tcp_0). The TCP service forms the test context.

Test architecture We let the testing tool play the role of up-to 3 clients. It interacts with the SBS implementation through the underlying TCP service, as depicted in Figure 5.8.

The model The model (in mCRL2) describes the behaviour of the SBS as a single-threaded server that, forever, accepts a message, and processes it by (a) updating the internal server state, and (b) (when appropriate) sending out one or more responses.

The model contains explicit actions to represent that a client connects to, or disconnects from, the server: ConnectRequest, resp. DisconnectRequest(id). In response to a ConnectRequest which is received from a client, the IUT will send a ConnectAcknowledge(id), where the id is the connection identification number (ID) chosen by the IUT. The ID is just a number (initial value 0) that the IUT increments for each received ConnectRequest. To close a connection immediately after creation, i.e. before an ConnectAcknowledge(id) is received, a client sends a DisconnectRequest(-1) (because the ID is not yet known). To close a connection for which the ID is known (say j), a client sends DisconnectRequest(j).

Like in the conference protocol model, for message parameters of a large domain, we represent services using an enumerated type with just a few (2) values.

The Adapter

We discuss the Adapter for i_2.

Ad 1: Interact with IUT We separately discuss Adapter and IUT startup (1A), connection handling (1B), and the actual interaction (1C).

1A: Startup Because this Adapter has to interact with the IUT over TCP, it needs to know the TCP address at which the IUT listens for connection requests.

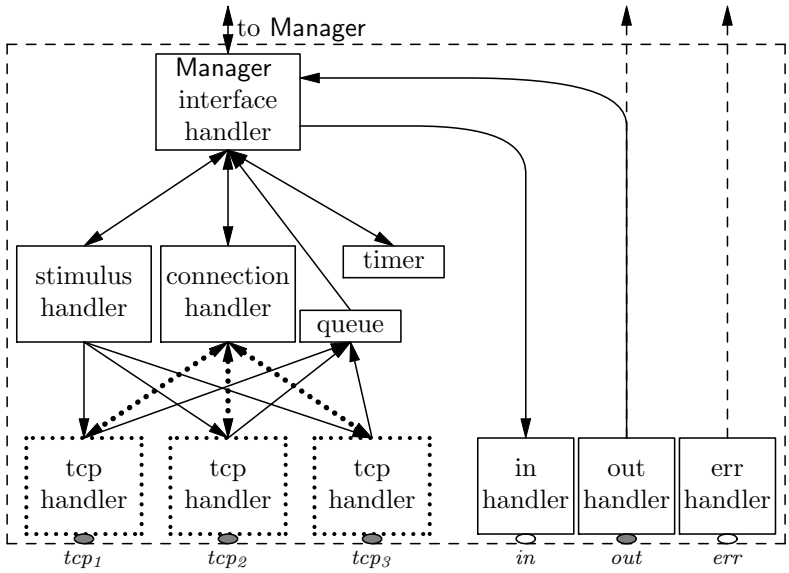


Figure 5.9: Software bus adapter

There are numerous ways to arrange the start up of Adapter and IUT, in such a way that this is achieved. Because IUT i_2 was created especially for testing, we could implement our own choice, which is as follows.

The Adapter starts the IUT, such that it can read the IUT's standard output and can write to the IUT's standard input, and then it waits for the first line of output from the IUT. Upon startup, the IUT starts a TCP listener, and lets the operating system decide on the TCP address (port number). If we choose a port number ourselves, we risk that it is already occupied. Then, it writes the resulting TCP address to its standard output, which the Adapter reads, and stores for subsequent use. Subsequent IUT output on its standard output and standard error is consumed, and logged for diagnostic purposes.

The Adapter will not write to the IUT's standard input, but at the end of a test run, it will close the IUT's standard input, which tells the IUT to terminate.

1B: Connection handling The Adapter dynamically opens and closes TCP connections to the IUT, in response to ConnectRequest and DisconnectRequest stimuli. When the Adapter opens a TCP connection, it starts a reader thread that reads messages written by the IUT and adds them to a shared observation queue. This queue has the same role as the queue in the Adapter for the toy implementation; but it lacks the built-in poll-until-a-timer-expires functionality.

The Adapter maintains a mapping between connection identification IDs, as used in the model labels and the corresponding connection handles that are used to write TCP messages. After opening a new connection, the Adapter gets to know the ID in the ConnectAcknowledge message which the SBS sends as soon as the connection has been created.

1C: Interaction The actual interaction between Adapter and SUT takes place in the TCP handler components.

For observations, interaction takes place in the reader threads that the Adapter starts for each connection that it opens. Two kinds of observations are possible. Either a message (model label) is obtained, or the Adapter observes that the IUT has closed the connection. Messages that are obtained, are added to the shared queue. When connection closure is observed, the connection handler component is informed, such that it can update its ID-to-connection mapping.

For all stimuli, except ConnectRequest and DisconnectRequest, the Adapter uses the ID in the label to look up the TCP connection handle, which it then uses to write the label onto the right TCP connection.

Ad 2: Mapping between labels and interactions For each kind of stimulus label, the Adapter has a function to handle the stimulus, and for each kind of observation, a function to handle the observation. The stimulus handling functions are invoked when a stimulus has to be applied; The observation handling functions are invoked to process an enqueued observation. Some of the stimulus handling functions open or close TCP connections; the other ones send a message (the label, in this case) over the connection that is indicated by the ID in the label. The observation handling function for the ConnectAcknowledge extracts the ID from the received label and updates the ID–client-handle mapping. The other observation handling functions are essentially a no-op: because the SBS i_2 implementation communicates using labels, no decoding is necessary.

Ad 3: Observing quiescence The Adapter is configured with a quiescence timeout value. During a test run, the Adapter remembers the time at which the last interaction with the SBS took place. If, when an observation is requested by the Manager, the observation queue is empty, the Adapter computes how long it must wait before it can report quiescence. If it does not have to wait (sufficient time passed since the last interaction with the SBS), an observation of quiescence is reported to the Manager immediately. Otherwise, a timer is started, and then, if an observation is added to the queue before the timer expires, the observation is reported to the Manager. Otherwise, an observation of quiescence is reported to the Manager.

Ad 4: Providing the Adapter interface The Adapter interface handler implements the Adapter interface functions.

Ad 5: Synchronisation Handling of tryStim is done in the same way as discussed in Section 5.1.2 for the CPE Adapter: The Adapter first checks for a pending observation, and, if one is obtained, returns the observation and discards the stimulus. Otherwise, the Adapter applies the stimulus by writing it to the right TCP connection handle. Once the Adapter has written the stimulus the stimulus is out of its control, and it reports that the stimulus has been applied.

5.2 Adapter Design

We now give a functional decomposition of the Adapter. We start with a high-level overview of the overall architecture (Section 5.2.1) which we subsequently refine in several steps.

In the high-level overview and the initial decomposition (Section 5.2.2) we implicitly assume a single point of interaction with the system under test (single PCO), and we ignore the concept of implementation access point (IAP) (i.e., single PCO and IAP coincide). We generalise this when we refine the decomposition (Section 5.2.3) and decompose the SUT into an IUT and a test context, like we have seen in the conference protocol example. In our last decomposition step (Section 5.2.4) we discuss support for dynamic changing interaction infrastructure, like we have seen in the software bus example.

5.2.1 High-level architecture overview

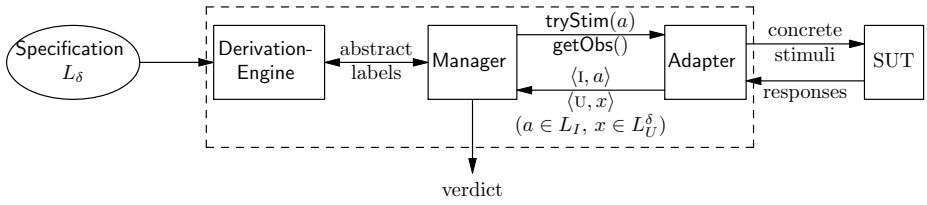


Figure 5.10: Figure 5.1 refined to show the Adapter interface.

In Figure 5.10 we depict the Adapter in our high-level architecture.

The Adapter interacts with the system under test by applying stimuli and obtaining observations. These stimuli and observations take place using the means of interaction that a given system under test offers to its environment. We refer to these as *concrete* interactions, and refer to data items exchanged in such interaction as *concrete* data. In the figure the stimuli travel from Adapter to system under test, and observations travel in the opposite direction, as a suggestion that they can be messages that are sent asynchronously: the Adapter (triggered by the Manager to do so) takes the initiative for the interactions that we refer to as stimuli, and the system under test takes the initiative for the interactions to which we refer as observations. For now we do not make additional assumptions about the interaction between Adapter and system under test.

The Adapter interacts with the Manager component using the *abstract* representations of the model from which the tests are derived: the labels in L_I (for stimuli) and in L_U^δ (for observations).

In the introduction to this chapter we listed common Adapter functionalities. We now refine the decomposition to see how this functionalities can be realised.

5.2.2 Initial decomposition step

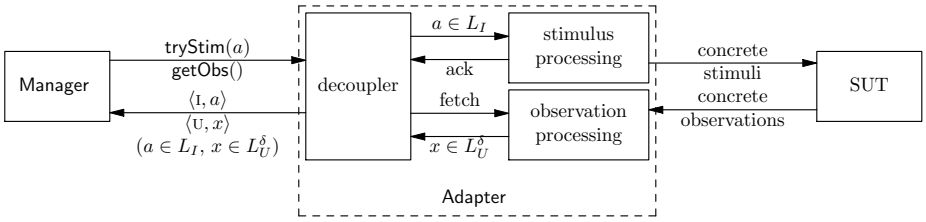


Figure 5.11: Initial decomposition of Adapter.

In Figure 5.11 we decomposed the Adapter into three components: (1) *decoupler*, (2) *stimulus processing*, and (3) *observation processing*. For now we just treat them as functional blocks, where decoupler and observation processing typically "run" concurrently.

The *decoupler* component has the same role as the Manager interface handler has in the examples (Figures 5.3, 5.5, 5.9): it provides the mapping between the synchronous interface between Adapter and Manager (where the Manager takes the initiative), and the asynchronous interface with the Adapter, where the Adapter takes the initiative to apply stimuli, but the SUT takes the initiative to produce responses. The *decoupler* deals with the case where a response, produced by the SUT, interferes with application of a stimulus.

The *stimulus processing* component is responsible for performing interactions that are initiated by the Adapter (i.e. stimuli) with the SUT. The *decoupler* takes the initiative (triggered by a tryStim request from the Manager) to request the *stimulus processing* component to apply a stimulus, and the *stimulus processing* component acknowledges application of a stimulus back to the *decoupler*.

The *observation processing* component is responsible for performing (or observing) interactions that are initiated by the SUT. The *decoupler* takes the initiative (triggered by a tryStim or getObs request from the Manager) to (try to) fetch observations from the *observation processing* component.

5.2.3 Refined decomposition

We now decompose the SUT into IUT and test context, refine the connection between Adapter and SUT, and refine the Adapter decomposition by decomposing the stimulus processing and observation processing components, thereby exposing the Adapter *state*, as depicted in Figure 5.12. We first discuss the SUT decomposition, and the refinement of the connection between Adapter and SUT, and then we discuss the refinement of the Adapter components.

SUT decomposition and Adapter–SUT connection refinement

We now decompose the SUT in a IUT and a test context, such that the Adapter may have to interact with the IUT via one or more underlying services that are part of the test context, and we allow both Adapter and IUT to interact

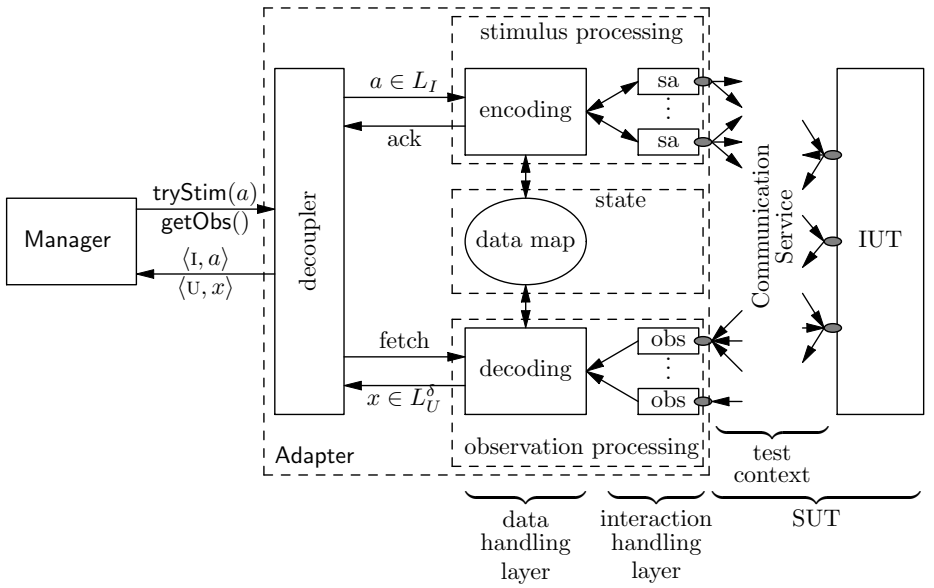


Figure 5.12: Refined decomposition of Adapter, introducing encoding and decoding components, and stimulus application (sa) and observer (obs) components. Back-arrows from sa components to encoding component are to ack that stimulus has been applied. (Note that, due to our choice for the placement of the sa and obs components, any PCO that is used both for stimuli and for observations, appears twice.) ($a \in L_I$, $x \in L_U^\delta$)

via multiple interaction points (PCOs resp. IAPs). In Fig. 5.12, as an example, the test context consists of a single underlying "Communication Service". In general, the Adapter may be able to access the IUT directly (in that case PCOs and IAPs coincide), and it may have to use one or more underlying services—in the conference protocol example of Section 5.1.2 we saw direct interaction at PCO/IAP cf_0, and the use of the underlying UDP service, at PCOs udp_1 and udp_2 to interact at IAP udp_0.

The underlying service may connect one PCO to multiple IAPs, and one IAP to multiple PCOs—also this we have seen in the conference protocol example. In such case, the underlying service must be provided (in addition to the data or messages that it must deliver) with address information, to be able to deliver at the right interface, and to be able to indicate to the recipient, who the originator was.

In Fig. 5.12 we do not show the underlying service address information (even though we do discuss it below); we do show it in Fig. 5.13 which we discuss in Section 5.2.4.

Adapter Components Decomposition

Inspiration for the decomposition of the stimulus processing and observation processing components comes from the two main Adapter functions: interacting

with the SUT, and mapping between these interactions and their representation as model labels. We decompose both components in a *data handling* and an *interaction handling* part. Together, the data handling parts form the *data handling layer*, and the interaction handling parts form the *interaction handling layer*.

The data handling layer consists of an *encoding* component—for stimulus processing, and a *decoding* component—for observation processing, together with a shared *data map*. The data map, which is part of the Adapter state, is used by both coding components. It contains mappings between abstract label values and concrete interaction values.

The interaction handling layer contains *stimulus application* (sa) components for stimulus processing, and *observer* (obs) components for observation processing. For each PCO at which the Adapter applies stimuli or obtains observations, the Adapter has an sa resp. an obs component.

In this decomposition we do not show the queue and timer components, which we have shown in the Adapter examples; we show these components in the detailed decomposition in Section 5.2.4.

Stimulus processing For stimuli, the encoding component maps model labels onto tuples $\langle sa, data, addr \rangle$ that contain the following elements:

sa an identification of the sa at which the interaction will take place;
data the concrete interaction data that has to be passed to the SUT;
addr optional addressing information for the underlying communication service, to identify an IAP, or to provide address information, that is not already associated with the *sa*, for a PCO.

Thus, after encoding, the *data* and *addr* elements are passed to the sa indicated by the *sa* element, to perform the actual interaction. When the interaction has taken place, an acknowledgement is given to the decoupler component, which then informs the Manager that the stimulus has been applied.

Observation processing For each interaction initiated by the SUT, the observation handler that handles the interaction creates a similar tuple $\langle obs, data, addr \rangle$ that contains

obs an identification of the observer component at which the interaction took place;
data the concrete interaction data that was passed by the SUT;
addr optional addressing information from the underlying communication service, that identifies the IAP, or that formed (part of) the identification of the PCO, and was not already associated with the *obs*.

The decoding component maps such tuples onto model labels. At this stage in the decomposition, we leave open what is queued: whether the observations, or the labels that result from decoding them.

5.2.4 Detailed decomposition

In this last decomposition step we

1. add a shared observation queue,

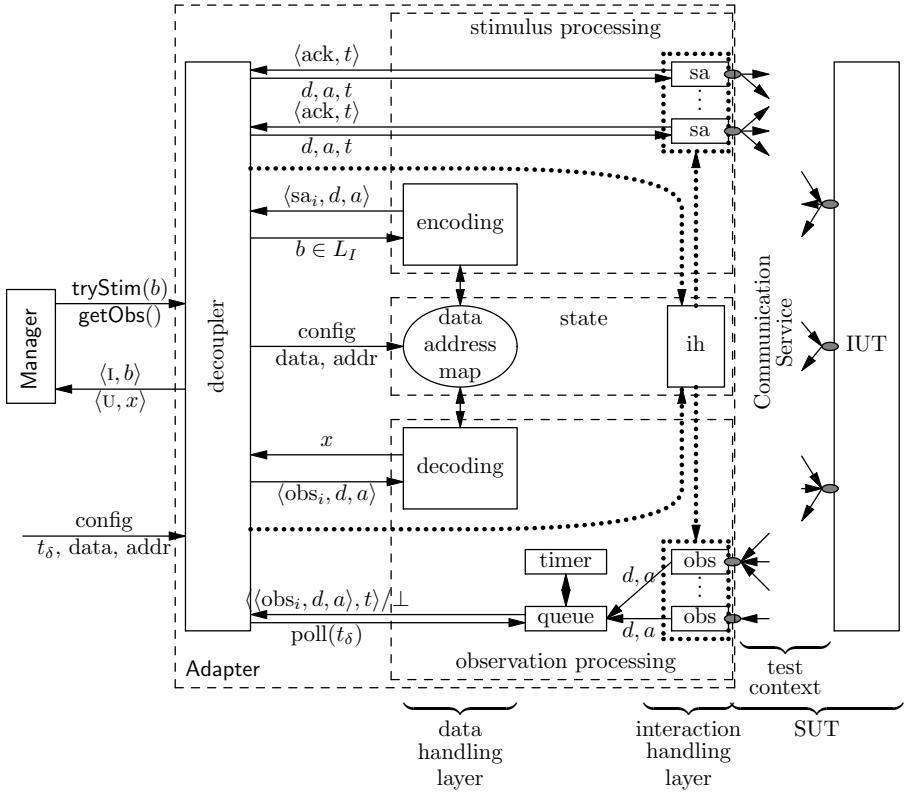


Figure 5.13: Detailed decomposition of Adapter, showing queue and timer components, and interaction handler (ih) that can create and delete sa and obs components, as it dynamically updates the interaction infrastructure. Also shown are underlying service data (d) and address information (a), and (optional) timestamps t associated with interactions. The "fetch" observation processing function of Fig. 5.12 has been replaced by "poll". (Note that, due to our choice for the placement of the sa and obs components, any PCO that is used both for stimuli and for observations, appears twice.) ($b \in L_I$, $x \in L_U^\delta$)

2. add a component that takes care of the dynamic reconfiguration of the interaction infrastructure, and
3. refine the interaction between the decoupler component and the stimulus- and observation processing components.

Ad 1: Shared observation queue In this decomposition we added the shared observation queue that we used in the first Adapter example (discussed in Section 5.1.1). The shared observation queue offers the same "poll" functionality as in that example: with the "poll" request, the decoupler gives a time-out value (t_δ in Fig. 5.13). If the queue contains one or more pending observations, or receives an observation within the time-out period, the earliest observation is returned; otherwise, special value $\perp$ is returned. A time-out value of 0 allows checking for pending observations.

Ad 2: Interaction handler component The interaction handler component ("ih" in Fig. 5.13) is there to adjust the interaction infrastructure in response to stimuli obtained from the Manager, and in response to certain observations (e.g., detection that a connection is closed by the SUT). It does this by interacting with existing sa and obs components in the Adapter (to reconfigure or delete them), and by creating new sa and obs instances.

Ad 3: Refined decoupler interaction In this decomposition the decoupler is the central component in the Adapter: it does not only interact with the Manager, but it also provides the link between the data handling components and the interaction handling ones. This allows the decoupler to check whether an observation has arrived while a stimulus was being encoded.

To handle a tryStim(b) request, the decoupler first checks whether the queue has a pending observation—if so, the pending observation is handed over to the decoding component, and the decoding result is returned to the Manager. Otherwise, the decoupler lets the encoding component encode b. This results either in a stimulus tuple $\langle sa_i, data, addr \rangle$, or in interaction infrastructure reordering data. Then, the decoupler again checks for a pending observation—if one is found, it is, after decoding, returned to the Manager. Otherwise, if the encoding result was a stimulus tuple, the stimulus data and (optional) address information are passed to right sa component; if the encoding result consists of interaction infrastructure reordering data, it is passed to the ih component.

To handle a getObs request, the decoupler polls the queue for an observation, using a time-out value that is computed using the configured time-out value t_δ, the timestamp of the last interaction with the SUT (the t elements in the results obtained from the sa and obs components), and the current time[3]. If an observation is obtained from the queue, it is decoded, and the decoding result is passed to the Manager. If $\perp$ is obtained from the queue, δ is passed to the Manager.

[3] We assume that the decoupler and the sa and obs components have access to a shared clock, which allows the sa and obs components to timestamp their interactions with the SUT, and which allows the decoupler to compute how much time has passed since the last such interaction. If no clock is present, the Adapter uses the configured time-out value t_δ.

This concludes our discussion of the Adapter design.

5.3 Summary and Related Work

In this chapter we described three Adapter instances, and an Adapter design that generalises upon these and other Adapter instances that we have worked on.

Below we mention related work.

TTCN-3 test system reference architecture The TTCN-3 [WDT+11, TTC] test system reference architecture (TSRA) contains functionality that serves the same purpose as our Adapter. The central element in the TTCN-3 TSRA is the TTCN-3 Executable, which results from compiling a suite of TTCN-3 tests. The TTCN-3 Executable corresponds, more or less, to the combination of test case, partition information and DerivationEngine in Fig. 4.10.

The TTCN-3 TSRA specifies two standardised interfaces between the TTCN-3 Executable and the other components in the TTCN-3 TSRA: the TTCN-3 Control Interface (TCI) and the TTCN-3 Runtime Interface (TRI).

The TCI interface connects the TTCN-3 Executable with a Test Management component, a Test Logging component, an Encoding and Decoding component, and a Component Handling component. The role of the Test Management component corresponds, more or less, to the role of our Manager in Fig. 4.10. In our design we do not have a separate component that corresponds to the TTCN-3 TSRA Test Logging component; our torx-explorer and torx-adapter interfaces contain support for logging, and the JTorX tool contains a logging module. The Encoding and Decoding component provides functionality that we placed in the data handling layer in our design. The Component Handling component provides functionality related to the use of parallel test components; our design has no such component: in our design, all interaction with the SUT takes place via functionality in the interaction handling layer.

The TRI interface connects the TTCN-3 Executable with a System Adapter and a Platform Adapter. The System Adapter contains functionality that corresponds to functionality in our interaction handling layer. The Platform Adapter provides the TTCN-3 system with a single notion of time—this, to a certain extent, corresponds to our timer component—and it contains the implementation of external TTCN-3 functions—we do not have a corresponding concept in our design.

A clear difference between the TTCN-3 setting and our LTS-based one, is that in the TTCN-3 setting much more information is readily available, like the interaction operation (message or procedure based), the interaction points, the types of the values exchanged in the interactions, etc.—in our setting this information is "hidden" in LTS labels.

Data abstraction Our encoding and decoding components use data and address mappings, which are part of the Adapter state. The concept of mappings that have state is taken to a next level in [AHK+12], which formalises a stateful Mapper component that maps between concrete and abstract values.

Behaviour abstraction Orthogonal to the mapping between concrete and abstract values lies the topic of action refinement, which studies the mapping between high-level abstract actions and the corresponding concrete behaviours [van11].

Chapter 6

Symbolic Extensions

In this chapter we discuss how we support requirement 6 of our design constraints: the tool should support very large and infinite state space models.

Although the approach that we discussed so far is based on the formalism of labelled transition systems (LTSs), in practice, when using our approach, models are typically not directly written directly as LTSs, but in a higher-level language, and a language-specific Explorer component provides the LTS "view" on a given model. As we have seen in Chapter 4, in this way we allow the use of modelling languages like mCRL2, Promela, and LOTOS. In languages like these, we can use variables to describe the system state, guards to specify whether certain behaviour is or is not enabled, and variables and expressions to describe the interaction between the system and its environment.

Although the expressivity of such languages enables construction of models of large systems, little of this expressivity remains in the LTS view on the model, as offered by the LTS-based Explorer components. This holds in particular for the description of interactions with the system environment. In a higher-level language model, we can describe concisely that in such interactions, any value of a given set of values may be exchanged. For example, we can describe an interaction as action $\mathsf{in}(x)$, with variable x a natural number, and a constraint (or guard) $0 \leq x < 5$. Because LTSs lack the concept of variables, however, an LTS has to contain a separate label (on a separate transition) for each possible combination of values of the variables in the interaction description. Thus, to represent above interaction description, we need five LTS labels: $\mathsf{in}(0), \mathsf{in}(1), \ldots, \mathsf{in}(4)$. With a small change to the constraint: $0 \leq x < 500$, the same interaction description maps onto 500 LTS labels, and with constraint $x \geq 5$, the LTS needs to contain an infinite number of labels!

Limitations of our LTS-based approach This "blow-up", which results from the construction of separate labels for each combination of possible values, causes two problems. Firstly, it may result in an LTS that contains states which have an infinite number of outgoing transitions—something that our LTS-based approach for test derivation cannot handle. Secondly, even when the LTS has a finite state space, there may be an "unbalance" in the number of

LTS labels necessary to represent different high-level interaction descriptions, and this unbalance may affect our approach of random on-line testing. For example, it may be the case that one high-level interaction description has a single corresponding LTS label, while another high-level interaction description, enabled together with the first one, has many corresponding LTS labels. During random on-line testing, it is much less likely that the label that corresponds with the former interaction description is chosen, than one that corresponds with the latter interaction description. In this way, the control flow of the higher-level model (two transitions to choose between) becomes obscured in the LTS mapping (many transitions to choose between, of which only one corresponds to one of the higher-level transitions, and all others to the other).

Symbolic Transition System We illustrate these problems, and our solution to them, using the *Symbolic Transition System* (STS) formalism of [FTW05, FTW06] as higher-level model. The STS formalism is a relatively simple modelling language, which nevertheless is rich enough to give rise to the above-mentioned phenomena. Our solution itself is not limited to the use of STS as higher-level modelling language, as we show at the end of this chapter, where we discuss how we can use our solution for timed testing. Instead, the STS formalism is just a convenient formalism to illustrate the problems and our solution. We use the STS formalism here, because it is well-known in the context of symbolic testing with **ioco**.

Our solution: Parameterised Transition Systems Our solution is to base the Explorer interface not on the LTS formalism, but on a slightly more expressive one, which we call *Parameterised Transition System* (PTS). We designed the PTS formalism for on-line testing from higher-level (i.e. symbolic, or timed) models.

During on-line testing, for each test step, the DerivationEngine is requested to compute the *potential behaviour* (the possible stimuli and expected responses) using interface functions in and out, after which the *actual behaviour* (interaction with the SUT) takes place, after which the test derivation state (the tester's view of where it "is" in the model) is updated using interface function next.

In the LTS-based approach, the potential behaviour is represented by the set of enabled LTS transitions, and the actual behaviour (of a correct SUT) by a subset of those transitions: namely, those transitions of which the label matches the label that represents the interaction with the SUT.

In contrast, in the PTS-based approach, the potential behaviour and the actual behaviour are represented by different transitions. The potential behaviour is represented by a set of enabled *parameterised* transitions. A parameterised transition has a *Parameterised* label, which may contain variables and a constraint over them. The idea is that each higher-level interaction description can be mapped onto a single PTS parameterised transition. Each destination state of a parameterised transition t has zero or more outgoing *instantiation* transitions (a.k.a. instantiations). Each instantiation i associates a value with each of the parameters of the original incoming transition t. Thus, where the LTS mapping maps a higher-level interaction description d onto n different LTS labels,

the PTS mapping maps d onto a single parameterised transition, together with n different instantiations. Effectively, we thus delay enumerating the combinations of possible values to the instantiation transitions. The actual behaviour is represented by one or more combinations of an enabled parameterised transition t and a subsequent instantiation i: those t,i combinations correspond to the LTS label of the actual behaviour when the values of i are substituted in the label of t.

In this way, a finite number of PTS labels in the Explorer and DerivationEngine interfaces can represent potential behaviour that would take an infinite number of LTS labels to represent. An important benefit of our solution is that on the PTS-based Explorer interface we only expose that information we want and need to expose. Just like we defined the LTS-based Explorer interface in terms of opaque LTS states, without exposing the contents of those states, we define the PTS-based Explorer interface in terms of opaque PTS states, and only expose parameters in parameterised labels, without exposing the contents of the PTS states—nor do we expose how the un-exposed PTS state information is updated when a transition in the higher-level language model takes place.

Remainder of this chapter In Section 6.1 we describe the STS formalism. In Section 6.2 we discuss two (running) examples that illustrate the two problems mentioned above. In Section 6.3 we present our solution: the parameterised transition system (PTS). In Section 6.4 we show how we can derive a PTS from an STS. In Section 6.5 we informally discuss the impact of using a PTS for testing, on our architecture. In Section 6.6 we extend our architecture to cope with PTSs. In Section 6.7 we sketch how we can use our PTS-based approach for timed testing.

6.1 Symbolic Transition System

6.1.1 Preliminaries

The STS definition uses basic concepts from first order logic to deal with data, for which we need to introduce additional notation. We combine the notation from [Eer94, FTW06, vS09].

Functions We denote a function f from A onto B as $f : A \to B$, where $\mathsf{dom}(f) = A$ and $\mathsf{cod}(f) = B$ denote the domain resp. codomain of f. We denote a partial function f from A onto B as $f : A \rightharpoonup B$, where $\mathsf{dom}(f)$ ($\subseteq A$) contains those $a \in A$ for which f is defined. For (partial) functions $f : B \rightharpoonup C$ and $g : A \rightharpoonup B$, we denote the composition of f and g as $f \circ g$ (which is defined on x if g is defined on x and f is defined on $g(x)$). Given a function f from A onto B, and $A' \subseteq A$, we denote the restriction of f to $A' \to B$ as $f\big|_{A'}$.

Sequence We use $\overline{x}$ to denote a sequence of x, i.e. $(x_1, \ldots, x_n)$. We use $|\overline{x}|$ to denote the number of elements of a sequence: $|(x_1, \ldots, x_n)| =_{\text{def}} n$. We

sometimes treat a sequence $(x_1, \ldots, x_n)$ as the set $\{x_1, \ldots, x_n\}$ when the context allows it.

Structure We assume a structure (Σ, M) with signature Σ and model M. Here $\Sigma = (\mathit{Sorts}, \mathit{Ops})$ is a many-sorted signature with Sorts a non-empty set of sorts, and Ops a set of function symbols (operations). Each $o \in \mathit{Ops}$ has a type $\in \mathit{Sorts}^n \times \mathit{Sorts}$, where $n \in \mathbb{N}$ is the arity of o. A corresponding model $M = ((U_s)_{s \in \mathit{Sorts}}, (f_o)_{o \in \mathit{Ops}})$ is a many sorted algebra. It contains for each sort $s \in \mathit{Sorts}$ of Σ a non-empty set U_s, which is called the universe of sort s, and for each operation $o \in \mathit{Ops}$ of Σ with $o : s_1 \times \ldots \times s_n \to s$ an interpretation $f_o : U_{s_1} \times \ldots \times U_{s_n} \to U_s$. We let $U = \biguplus_{s \in \mathit{Sorts}} U_s$.

We assume that Sorts contains a boolean sort bool, and that Ops contains the usual boolean operations.

We assume a global universe of sorted variables Vars, and we use subsets $X, Y \subseteq \mathit{Vars}$. We use Vars^s to denote the set of variables of sort s.

Terms Assuming a many-sorted signature Σ we use T_Σ $(= T_\Sigma(\emptyset))$ to denote the set of ground terms that can be constructed from Σ, and $T_\Sigma(X)$ to denote the set of well-typed terms over a set of variables X.

We use T_Σ^s to denote the set of terms of sort s; for instance, the set of all boolean expressions over X is denoted as $T_\Sigma^{\mathsf{bool}}(X)$.

We use $\mathsf{sort}(t)$ to denote the sort of a term $t \in T_\Sigma(\mathit{Vars})$. We use $\mathsf{var}(t)$ to denote the set of variables that occur in a term t.

We assume that each element of U has a canonical term representation as a term in normal form.

Term-mapping and substitution We refer to a partial function $\sigma : X \rightharpoonup T_\Sigma(Y)$ as a term-mapping. The identity term-mapping id is defined as $\mathsf{id}(x) = x$ for all $x \in \mathit{Vars}$. We use $\{x \leftarrow t_x, y \leftarrow t_x, \ldots\}$ to denote a term-mapping that maps x onto t_x, y onto t_y, etc. We use σ_X to denote the restriction of σ that is only to be applied on variables from X, i.e.

$$\sigma_X(x) = \begin{cases} \sigma(x) & \text{if } x \in X \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Given a term $t \in T$, we use $t[\sigma]$ to denote the substitution in t of $\sigma(x)$ for every $x \in \mathsf{var}(t)$. We also use $\overline{x \leftarrow E}$ to denote a particular substitution where each variable x_i must be replaced by the term E_i (and all variables not in $\overline{x}$ are unaffected). We use $\mathsf{dom}(\sigma)$ to denote the set of variables that are substituted (i.e. $\mathsf{dom}(\overline{x \leftarrow E}) =_{\text{def}} \overline{x}$). We use $\mathsf{terms}(\sigma)$ to denote the terms that are substituted for these variables (i.e. $\mathsf{terms}(\overline{x \leftarrow E}) =_{\text{def}} \overline{E}$). To denote the set of variables used in the images of σ we use $\mathsf{var}(\sigma) =_{\text{def}} \mathsf{var}(\mathsf{terms}(\sigma))$.

Valuation A valuation is a function $\vartheta : \mathit{Vars} \to U$, with $\vartheta(\langle x_1, \ldots, x_n \rangle) = \langle \vartheta(x_1), \ldots, \vartheta(x_n) \rangle$ for a given tuple of variables $\langle x_1, \ldots, x_n \rangle$ with $x_i \in \mathit{Vars}, 1 \leq i \leq n$. Let $*^s$ denote a fixed arbitrary element of the set U_s. A partial valuation

is a partial function $\vartheta_X : \mathit{Vars} \rightharpoonup U$ with $\mathsf{dom}(\vartheta_X) = X$; ϑ_X can be extended to a total valuation as follows:

$$\vartheta(x) = \begin{cases} \vartheta_X(x) & \text{if } x \in X \\ *^s \in U^s & \text{if } x \in \mathit{Vars}^s \setminus X \end{cases}$$

Having two partial valuations $\vartheta, \varsigma : \mathit{Vars} \rightharpoonup U$ with $\mathsf{dom}(\vartheta) \cap \mathsf{dom}(\varsigma) = \emptyset$, their union $(\vartheta \cup \varsigma) \in \mathit{Vars} \rightharpoonup U$ is defined as

$$(\vartheta \cup \varsigma)(x) = \begin{cases} \vartheta(x) & \text{if } x \in \mathsf{dom}(\vartheta) \\ \varsigma(x) & \text{if } x \in \mathsf{dom}(\varsigma) \end{cases}$$

The extension of a valuation ϑ to terms is called a term-evaluation and denoted $\vartheta_{\mathsf{eval}} : T_\Sigma(\mathit{Vars}) \to U$.

The satisfaction of a boolean expression φ w.r.t. a given valuation ϑ is denoted $\vartheta \models \varphi$; this is equivalent to $\vartheta_{\mathsf{eval}}(\varphi) = \mathsf{true}$.

6.1.2 Syntax and semantics of Symbolic Transition System

The following definition of a Symbolic Transition System is based on [FTW06]; we changed notation to match the previous section, and made three other changes as indicated in the bibliographic note below.

In the definition we use the concept of a gate; we assume a global universe of gates *Gates*.

Definition 6.1.1

A *Symbolic Transition System* (STS) is a tuple $S = \langle L, l_0, \mathcal{V}_L, \mathcal{I}, \mathcal{G}, \rightarrow \rangle$, where

- L is a set of locations,
- l_0 is the initial location,
- $\mathcal{V}_L : L \to \mathcal{P}(\mathit{Vars})$ associates a set of location variables with every location; we also use $\mathcal{V} =_{\text{def}} \bigcup_{l \in L} \mathcal{V}_L(l)$,
- $\mathcal{I} \subseteq \mathit{Vars}$ is a set of interaction variables; $\mathcal{V} \cap \mathcal{I} = \emptyset$, and $\mathsf{Var} =_{\text{def}} \mathcal{V} \cup \mathcal{I}$.
- $\mathcal{G} \subseteq \mathit{Gates}$ is the set of gates; constant $\tau \notin \mathcal{G}$ denotes an unobservable gate; $\mathcal{G}_\tau$ abbreviates $\mathcal{G} \cup \{\tau\}$. Every gate $g \in \mathcal{G}_\tau$ has a type $\in \mathit{Sorts}^n, n \in \mathbb{N}$, denoted as $\mathsf{type}(g)$, where we refer to n as the arity of the gate, denoted $\mathsf{arity}(g)$. $\mathsf{arity}(\tau) = 0$[1]. $\mathsf{params}(g) \in \mathcal{I}^n$ yields a tuple of size $\mathsf{arity}(g) = n$ of distinct interaction variables of type $\mathsf{type}(g)$.
- $\rightarrow$ is the switch relation, where each $t \in \rightarrow$ is a tuple $\langle l, g, \varphi, \rho, l' \rangle$ where
  - $l \in L$ is the source location,
  - $l' \in L$ is the destination location,
  - $g \in \mathcal{G}_\tau$ is a gate,
  - $\varphi \in T_\Sigma^{\mathsf{bool}}(\mathcal{V}_L(l) \cup \mathsf{params}(g))$ is the switch restriction, which must hold for the switch to be enabled, and

[1] Also in [FTW06] $\mathsf{arity}(\tau) = 0$. We are aware that this imposes a limitation: for example, full LOTOS [ISO89] cannot be expressed as STS; we leave that for future work.

$\rho : \mathcal{V}_L(l') \to T_\Sigma(\mathcal{V}_L(l) \cup \mathsf{params}(g))$ is the update mapping, which associates new values with the location variables of the destination location when the switch is traversed.

We use the following functions and vocabulary:

1. $\langle S, \iota \rangle$ is an initialised STS S, where $\iota : \mathcal{V}_L(l_0) \to U$ initialises all variables associated with l_0.

We write $l \xrightarrow{g,\varphi,\rho} l'$ for $\langle l, g, \varphi, \rho, l' \rangle \in \to$.

□

The interpretation of an STS is defined by defining a mapping onto an LTS.

Definition 6.1.2
Let $S = \langle L, l_0, \mathcal{V}_L, \mathcal{I}, \mathcal{G}, \to \rangle$ be an STS, and $\langle S, \iota \rangle$ an initialised STS. Its interpretation is defined as the LTS $[\![S]\!]_\iota = \langle (L \times (\mathcal{V} \to U)), (l_0, \iota), A, T \rangle$ for all $\iota \in (\mathcal{V} \to U)$, where

$A = \bigcup_{g \in \mathcal{G}} (\{g\} \times U^{\mathsf{arity}(g)})$ is the set of actions, and
$T \subseteq (L \times (\mathcal{V} \to U)) \times (A \cup \{\tau\}) \times (L \times (\mathcal{V} \to U))$ is defined by the following rule:

$$\frac{l \xrightarrow{g,\varphi,\rho} l' \qquad \varsigma \in (\mathsf{params}(g) \to U) \qquad \varsigma \text{ is well-typed} \qquad \vartheta \cup \varsigma \models \varphi \qquad \vartheta' = (\vartheta \cup \varsigma)_{\mathsf{eval}} \circ \rho}{(l, \vartheta) \xrightarrow{(g, \varsigma(\mathsf{params}(g)))} (l', \vartheta')}$$

The semantics of an initialised STS $\langle S, \iota \rangle$ is given by the LTS $[\![S]\!]_\iota$.

□

In Section 6.2, below, we give two example STS models.

Bibliographical note The STS definition, and its interpretation, given as Definition 6.1.1 resp. 6.1.2 are based on [FTW06]. Our definitions differs from those in [FTW06] in the following aspects: (1) we use boolean expressions as switch restriction, where [FTW06] allows first order formulas; (2) where we associate a "local" set of location variables with each location, [FTW06] only has a global set of location variables; (3) we renamed function type to params.

6.2 Motivating Examples

We now give two example STS models, one of a simple music player, and one of a two-slot buffer. We use these models to illustrate the two problems that we mentioned at the start of this chapter. The two-slot buffer LTS has states with an infinite number of outgoing transitions, and the music player LTS has an unbalance in the number of LTS transitions that are generated for the switches of its STS model.

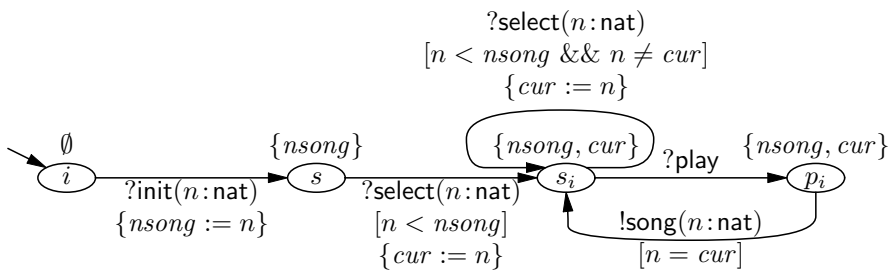


Figure 6.1: STS model of music player. With each state, its associated set of state variables is shown. Switch restrictions appear enclosed between square brackets "[" and "]", and update mappings between braces "{' and "}". Switch restrictions **true** and update mappings id are not shown.

6.2.1 Music Player

Imagine a music player with the following interface. After being initialised with a number of songs, it presents a (finite, but possibly very long) list of these songs, and a play button. The user can select a song from the list. Once the user has selected a song, the play button becomes enabled, and the user can play the selected song, or select a different song. While the song is playing, no buttons are enabled. We use natural numbers $0, 1, \ldots,$ *nsong–1*, to identify (represent) the *nsong* songs. The STS is shown in Fig. 6.1, a partial LTS in Fig. 6.2.

STS The music player STS structure has the following elements:

L is $\{i, s, s_i, p_i\}$,
l_0 is i,
$\mathcal{V}_L$ is $(i \mapsto \emptyset,\ s \mapsto \{nsong\},\ s_i \mapsto \{cur, nsong\},\ p_i \mapsto \{cur, nsong\})$,
$\mathcal{V}$ is $\{cur, nsong\}$, $cur \in Vars^{\mathsf{nat}}$, $nsong \in Vars^{\mathsf{nat}}$,
v_0 is $\emptyset$,
$\mathcal{I}$ is $\{n\}$, $n \in Vars^{\mathsf{nat}}$,
G is $\{?\mathsf{init}, ?\mathsf{select}, ?\mathsf{play}, !\mathsf{song}\}$,
T is $\{\ i \xrightarrow{?\mathsf{init},\mathsf{true},nsong \leftarrow n} s,$
$s \xrightarrow{?\mathsf{select},n<nsong,cur \leftarrow n,nsong \leftarrow nsong} s_i,$
$s_i \xrightarrow{?\mathsf{select},n<nsong \&\& n \neq cur,cur \leftarrow n,nsong \leftarrow nsong} s_i,$
$s_i \xrightarrow{?\mathsf{play},\mathsf{true},\mathsf{id}} p_i,$
$p_i \xrightarrow{!\mathsf{song},n=cur,\mathsf{id}} s_i\ \},$

and the params function is defined as follows:

- $\mathsf{params}(?\mathsf{init}) = \mathsf{params}(?\mathsf{select}) = \mathsf{params}(!\mathsf{song}) = (n)$,
- $\mathsf{params}(?\mathsf{play}) = ()$.

The STS uses two location variables: *cur* refers to the selected song, *nsong* contains the number of songs available.

The switch labelled ?init(n : nat) from initial location i to location s represents the initialisation. The switch labelled ?select(n : nat)[$n < nsong$] from location

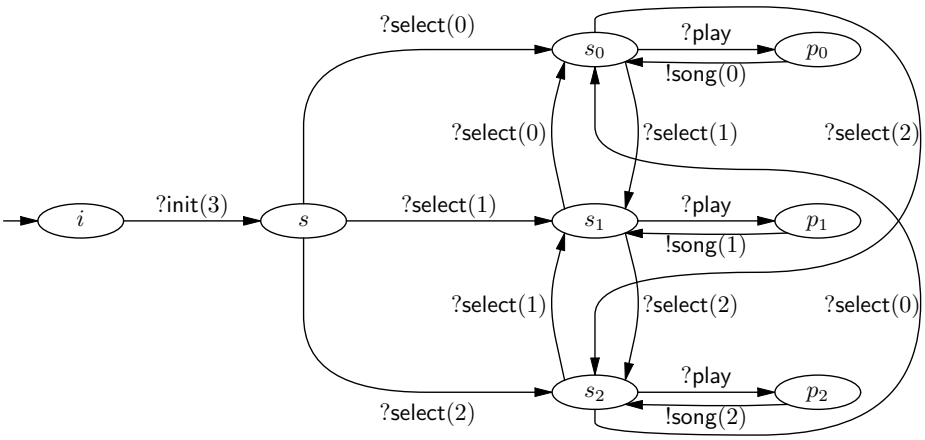


Figure 6.2: Labelled transition system of the music player for the case where it is initialised with 3 songs (i.e. $nsong = 3$). Note that this displays only part of the entire (infinite) LTS. In particular, the initial state has (infinitely) many more outgoing transitions than the sole one drawn here.

s to s_i represents the selection of song n: this switch updates variable cur to n. The switch labelled ?play from location s_i to p_i represents pressing the play button; playing of the selected song is represented by the switch labelled !song$(n : \mathsf{nat})[n = cur]$ back to s_i. In that location we can also select any song that is not the currently selected one (cur), as represented by the self-loop labelled ?select$(n : \mathsf{nat})[n < nsong \;\&\&\; n \neq cur]$.

LTS In Figure 6.2 we show the LTS of the music player with an initialisation of 3 songs, i.e. for the case where $nsong = 3$. From the initial state i we have a transition labelled ?init(3) to state s. This transition represents the initialisation. From state s we have for each song i a transition labelled ?select(i) to a state s_i in which song i is selected. In each state s_i we can press the play button (represented by a transition labelled ?play) which brings us to a corresponding state p_i. In state p_i we can hear song i (represented by transitions labelled !song(i)) and then return to state s_i. Moreover, in each state s_i we can select any song j that is not the currently selected one (i), as represented by transitions labelled ?select(j) for each $j \neq i$ from each state s_i to a state s_j.

So, for the general case, using $nsong$ as a named constant, we have transitions with the following action labels:

?init($nsong$)	?select(0)	?play	!song(0)
	?select(1)		!song(1)
	...		...
	?select($nsong–1$)		!song($nsong–1$)

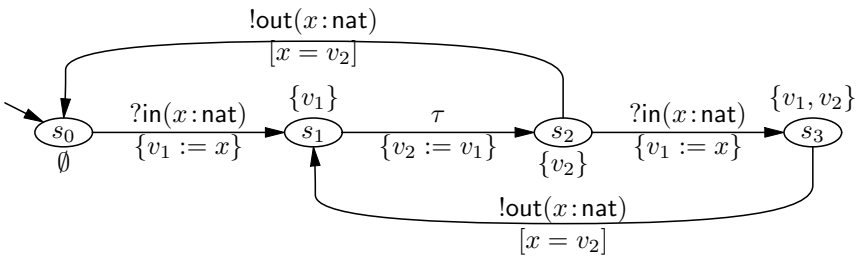


Figure 6.3: STS model of two-slot buffer.

STS vs. LTS comparison In the STS model, location s_i has two outgoing switches, one with action ?select, and one with ?play; both switches are enabled when the music player is initialised with at least two songs (i.e. $nsong > 1$). In the LTS, after an initialisation with $nsong$ songs ($nsong > 1$), each of the states s_i ($0 \leq i < nsong$) has $nsong$ outgoing transitions: a single outgoing transition labelled ?play, and $nsong$–1 outgoing transitions that ?select another song. This demonstrates the unbalance in the mapping from STS transitions to LTS transitions that we mentioned in the introduction: two STS switches are mapped onto in total $nsong$ LTS transitions, where one switch is mapped onto 1 of the transitions, and the other onto the other $nsong$–1 transitions. The larger the value of $nsong$ is, the more the control flow of the STS is distorted in the corresponding LTS.

6.2.2 Two-slot Buffer

Imagine a buffer that consists of two cells. We store natural numbers in the cells, i.e. we insert them into the cells, and take them out again, in FIFO order. A number that is inserted, is stored in the first cell. When the second cell is empty, and the first cell contains a number, the number is transferred to the second cell—this is an internal transition in the buffer. When the second cell is non-empty, its number can be taken out.

STS An STS model is depicted in Figure 6.3. Location variables v_1 and v_2 represent the buffer cells. The model has 5 switches. The two switches labelled ?in(x : nat) represent putting a number (x) into the buffer; this puts the number in the first cell. The two switches labelled !out(x : nat) represent getting a number from the buffer; this takes the number from the second cell. The switch labelled τ represents the internal transition where the value in the first cell is put (copied) into the second cell. Note that for the initial location we only have a single switch with label ?in(x : nat).

LTS A small part of the state space of the model is depicted in Figure 6.4. Enumeration of the outgoing transitions of the initial state gives us an infinite number of transitions with labels like ?in(0), ?in(1), ?in(2), etc. Such infinite enumeration cannot be handled by our LTS-based approach. On the other

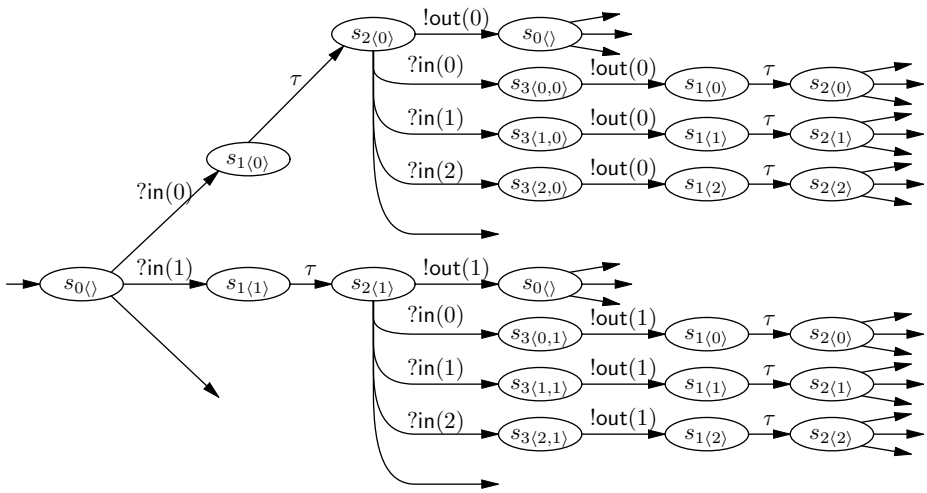


Figure 6.4: Initial part of labelled transition system of the two-slot buffer. The state labels show the contents of the cells between angled brackets, and a state number (s_0 etc.) that refers to the state numbers in the symbolic model that is given in Figure 6.3. The transitions labelled with τ transfer (copy) the contents of the first cell to the second cell. Of some states we have drawn multiple (equivalent) copies (as is visible in the state labels), to avoid a more complicated non-planar graph.

hand, as shown in Fig. 6.3, our symbolic model is perfectly able to represent this infinite behaviour with only four locations and five switches.

6.3 Parameterised Transition System

We now discuss the *parameterised transition system* (PTS) formalism. We first define PTS syntax and semantics, after which we discuss the PTS for the two-slot buffer example.

The PTS formalism was designed as a representation of a symbolic (e.g. STS) model that fits the two activities of our on-line testing approach: the exploration of potential behaviour as well as the execution of actual behaviour. In this sense, we use a PTS to give access to the symbolic model represented by the PTS. A PTS combines a symbolic representation of the potential behaviour with a non-symbolic representation of the actual behaviour. In the symbolic representation it only exposes (in terms of STS concepts) interaction variables and switch restrictions of the original model, without exposing its state variables or update mappings. A PTS may contain sequences of two or more parameterised transitions, without intermediate instantiation, i.e. both fully symbolic models and LTSes are just special cases of such PTS formalism. However, for our on-line testing approach, we do not need such generality. A restricted form of the PTS, the *alternating PTS* (APTS), suffices for our on-line testing approach: it is a

perfect fit for the strict alternation between exploration of potential behaviour and execution of actual behaviour. Therefore, to keep the definitions simple, we restrict ourselves, and only define the *alternating* PTS formalism. As we will see, we can describe the APTS formalism as a special kind of STS.

By replacing the LTS-based interfaces in our test derivation architecture by (A)PTS-based ones, we overcome the problems mentioned in the introduction and illustrated in Section 6.2.

PTS edges: transitions and instantiations A PTS has two kinds of edges: transitions and instantiations. As we will see, the potential behaviour is represented using transitions, which can be parameterised essentially like STS transitions. When we want to stress that a transition has no parameters, we refer to it as an *instantiated* transition. Actual behaviour is represented by the combination of a parameterised transition t and a subsequent instantiation i that associates values with the parameters of t.

In our figures, we use the convention that parameterised transitions are denoted by dashed edges, and instantiated transitions are denoted using continuous edges; instantiations are drawn as dotted edges.

PTS nodes: states Like STS locations, PTS locations have associated sets of variables. These are regarded as instantiation obligations.

In our figures, we use the convention that parameterised states (i.e. states with a non-empty set of location variables) are drawn with dashed borders, and instantiated states are drawn with solid borders.

We now first, in Section 6.3.1, define the alternating PTS formalism; then, we give an example in Section 6.3.2.

6.3.1 APTS Syntax and Semantics

We define a PTS as an STS that is extended with instantiation transitions, and that has restrictions on the location variables and the update mappings.

We use the location variables to keep track of the instantiation obligation, i.e. of those interaction variables that still need to be instantiated. Therefore, in a PTS we need, for each interaction variable $x \in \mathcal{I}$, a corresponding location variable $x_L \in \mathit{Vars}$, and we use the update mapping ρ to state this correspondence.

Definition 6.3.1
An *alternating parameterised transition system* (APTS) is a tuple $\langle L, l_0, \mathcal{V}_L, \mathcal{I}, \mathcal{G}, \rightarrow, \Phi, \cdots\!\!\succ \rangle$ where

$S = \langle L, l_0, \mathcal{V}_L, \mathcal{I}, \mathcal{G}, \rightarrow \rangle$ is an STS,
$\Phi \subseteq (L \rightarrow T_\Sigma^{\mathsf{bool}}(\mathsf{Var}))$ associates a constraint with each location, and
$\cdots\!\!\succ$ is an instantiation relation, where each $i \in \cdots\!\!\succ$ is a tuple $\langle l, b, l' \rangle$ where
l is the source location,
l' is the destination location, and
$b \in (\mathcal{V}_L(l) \mapsto T_\Sigma)$ is a binding (i.e., a term-mapping to ground terms),

and where
i-1. $\mathsf{dom}(b) \neq \emptyset$ and $\mathsf{dom}(b) \subseteq \mathcal{V}_L(l)$,
i-2. $\mathcal{V}_L(l') = \mathcal{V}_L(l) \setminus \mathsf{dom}(b)$,
i-3. $b \cup \sigma \models \Phi(l)$ for some σ, and
i-4. $\Phi(l') = \Phi(l)[b]$.
and where
1. $\mathcal{V}_L(l_0) = \emptyset$, and
2. $\Phi(l_0) = \emptyset$.
We impose the following additional restrictions on each transition $l \xrightarrow{g,\varphi,\rho} l' \in \rightarrow$:
t-0. $\mathcal{V}_L(l) = \emptyset$,
t-1. $\mathcal{V}_L(l') = \{x_L \mid x \in \mathsf{params}(g)\}$,
t-2. $\mathsf{cod}(\rho)$ is restricted to $\mathsf{params}(g)$,
t-3. ρ is a type-correct bijection,
t-4. $\rho\big|_{\mathcal{V}_L(l)} = \mathsf{id}\big|_{\mathcal{V}_L(l)}$, and
t-5. $\Phi(l') = \varphi$.

□

For each transition $l \xrightarrow{g,\varphi,\rho} l' \in \rightarrow$, the update mapping $\rho : (\mathcal{V}_L(l') \rightarrow \mathsf{params}(g))$ associates each interaction variable $x \in \mathsf{params}(g)$ with a corresponding variable $x_L \in \mathcal{V}_L(l')$. In the instantiation transitions $i = \langle l, b, l' \rangle \in \cdots\!\!\succ$ we associate values with the location variables of the source location, i.e. with variables x_L.

We write $l \overset{b}{\cdots\!\!\succ} l'$ for $\langle l, b, l' \rangle \in \cdots\!\!\succ$, and $l \overset{b}{\cdots\!\!\succ}$ if $\exists l' : \langle l, b, l' \rangle \in \cdots\!\!\succ$. Because ρ has such a regular form in a PTS, we typically omit it and write $l \xrightarrow{g,\varphi} l'$ for $\langle l, g, \varphi, \rho, l' \rangle \in \rightarrow$, and $s \xrightarrow{g,\varphi}$ if $\exists l' : \langle l, g, \varphi, \rho, l' \rangle \in \rightarrow$.

Well-definedness An APTS is well-defined when it fulfils the following consistency constraints, see Figure 6.5:

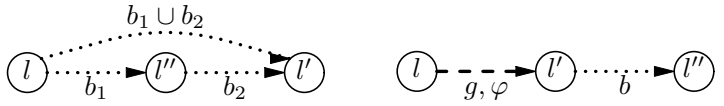


Figure 6.5: APTS well-definedness consistency constraints 1 (a) and 2 (b)

1. if $l \overset{b_1}{\cdots\!\!\succ} l'' \overset{b_2}{\cdots\!\!\succ} l'$, then also $l \overset{b_1 \cup b_2}{\cdots\!\!\succ} l'$
2. if $l \overset{g,\varphi}{\cdots\!\!\succ} l' \in \rightarrow$ and $\mathcal{V}_L(l') \neq \emptyset$ and $b \cup \sigma \models \varphi$ for some σ then $\exists l' \cdot \overset{b}{\cdots\!\!\succ} l''$

Ad constraint 1: This constraint says that when two instantiations can be applied separately, one after the other, then there is also a 'shortcut' instantiation in which both instantiations are applied together.

Ad constraint 2: This constraint says that from a parameterised state, there only is no outgoing instantiation when the constraint on the incoming transition does not hold.

In the remainder we only consider well-defined alternating PTSs.

Semantics of an APTS The interpretation of a well-defined APTS is defined by defining a mapping onto an LTS.

Definition 6.3.2
Let $P = \langle L, l_0, \mathcal{V}_L, \mathcal{I}, \mathcal{G}, \rightarrow, \Phi, \cdots\!\!\succ \rangle$ be an APTS, and $\iota : \mathcal{V}_L(l_0) \rightarrow U$ an initialisation that initialises all variables associated with l_0. Its interpretation $[\![P]\!]_\iota$ is defined as the LTS $[\![P]\!]_\iota = \langle S_{LTS}, s_0, A, \rightarrow \rangle$, where
$S_{LTS} = \{S \mid \mathcal{V}_L(S) = \emptyset\}$ is the set of states, and
$s_0 = \begin{cases} l_0 & \text{if } \mathcal{V}_L(l_0) = \emptyset \\ l' & \text{if } \mathcal{V}_L(l_0) \neq \emptyset \wedge l_0 \cdot^{\iota} \!\!\succ l' \end{cases}$
$A = \bigcup_{g \in \mathcal{G}} (\{g\} \times U^{\mathsf{arity}(g)})$ is the set of actions, and
$\rightarrow \subseteq (S_{LTS} \times (A \cup \{\tau\}) \times S_{LTS})$ is defined by the following rules:

$$\frac{l \xrightarrow{g,\varphi,\rho} l' \qquad \mathcal{V}_L(l) = \mathcal{V}_L(l') = \emptyset \qquad \models \varphi}{s \xrightarrow{g,()} s'}$$

$$\frac{\begin{array}{ccc} l \xrightarrow{g,\varphi,\rho} l' & \mathcal{V}_L(l) = \emptyset & \mathcal{V}_L(l') \neq \emptyset \\ l' \cdot^{b} \!\!\succ l'' & \mathcal{V}_L(l'') = \emptyset & \varsigma = b \circ \rho^{-1} \qquad \varsigma \models \varphi \end{array}}{l \xrightarrow{g,\varsigma(\mathsf{params}(g))} l''}$$

□

6.3.2 Example: APTS of two-slot buffer

In Figure 6.6 we show the initial part of an APTS for the two-slot buffer example.

With each state, we show the values stored in the two buffer slots, as far as the variables that hold them are "visible" in the state. When a parameterised transition has taken place, and the concrete interaction value is not yet known, we show the parameter (e.g. x). Note that we only show these values to make it easier to relate the APTS to the STS and the LTS; these values are not part of the formal APTS definition.

From the initial state ($s_{0<>}$) it shows a parameterised transition, labelled $?\mathsf{in}(x : \mathsf{nat})$, to $s_{1<x>}$. This represents the initial potential behaviour. For the actual value of x in the first interaction with the tester, there is an infinite number of possibilities, as represented by the infinite number (only two have been drawn) of outgoing instantiation edges—each with a different value for parameter x—from state $s_{1<x>}$. When we compare this with the STS model and the LTS, then, instead of a single symbolic transition as in the STS, or an infinite number of LTS transitions, we now have first a symbolic transition followed by an infinite number of instantiations.

From each of the states reached by one of the instantiations, only a τ-labelled transition is possible. Note that this same transition is also present in the STS and the LTS of resp. Fig. 6.3 and 6.4.

From each of the states reached by one of these τ-labelled transitions, there is one parameter-free transition, and one parameterised transition. The parameter-free transition represents taking out the first value from the buffer—the value is

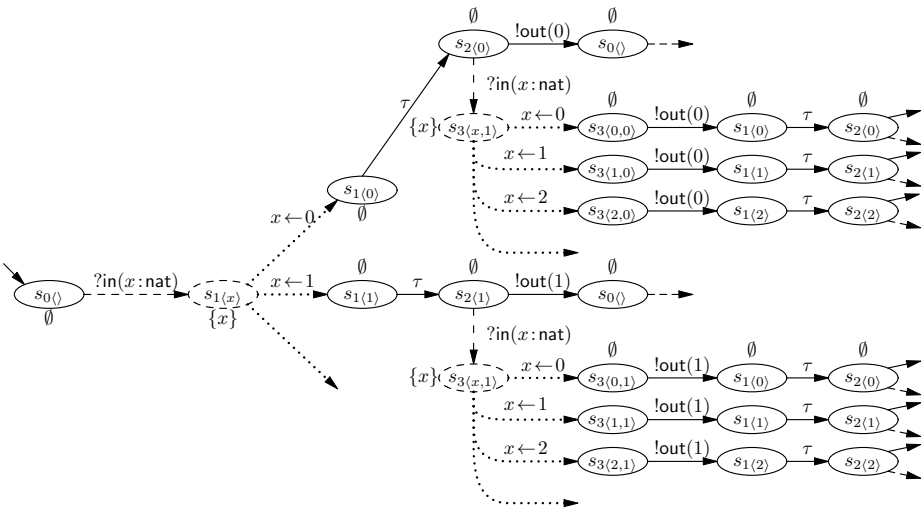


Figure 6.6: Initial part of APTS for the two-slot buffer of Figures 6.3 and 6.4, showing parameterised states (dashed border), parameter-free states and instantiated states (solid border), parameterised transitions (dashed edges), parameter-free transitions (solid edges), and instantiations (dotted edges). With the states, we indicate the instantiation obligation. We have drawn edges like $l \xrightarrow{\mathsf{!out}(0)} l'$ for transition-instantiation pairs where the switch restriction of the transition restricts all parameters to a single value, i.e. where we have a transition $l \xrightarrow{\mathsf{!out}(v:nat)[v=0]} l''$ and a corresponding instantiation $l'' \cdots\!\overset{v\leftarrow 0}{\cdots}\!\cdots\!\triangleright\, l'$.

known, because it was given in the instantiation, and thus there is no need for a parameter. The parameterised transition represents inserting one more value into the buffer; the value is still unknown, hence the parameter, which again is instantiated via one of the instantiation edges that follow the parameterised transition.

6.4 Derivation of APTS from STS

To illustrate the use of an APTS to give access to a higher-level model, we show how we can use a APTS to give access to an STS model. First, in Section 6.4.1, we describe the STS-to-APTS mapping, and then, in Section 6.4.2, we give an example: we show the use of this mapping on the music player STS model from Section 6.2.1.

6.4.1 Mapping STS to APTS

We define the mapping from an initialised STS $\langle S, \iota\rangle$, to an alternating PTS P below. We first describe the APTS states, transitions, and the location variables and update mapping. We formalise the mapping in Definition 6.4.1.
Below we assume that STS is given as $S = \langle L_S, l_{0,S}, \mathcal{V}_{L,S}, \mathcal{I}_S, \mathcal{G}_S, \rightarrow_S\rangle$, with

$\mathcal{V}_S \;=_{\text{def}} \bigcup_{l \in L_S} \mathcal{V}_{L,S}(l)$, and
$\mathsf{Var}_S \;=_{\text{def}} \mathcal{V}_S \cup \mathcal{I}_S$.

APTS States Given STS S, we construct APTS states as tuples $\langle l, X, \vartheta, \varphi, \rho \rangle$ with
$l \in L_S$ a location of S;
$X \in \{\emptyset\} \cup \{\mathcal{V}_g \mid g \in \mathcal{G}_S\}$ an instantiation obligation (set of state variables that correspond to the parameters of incoming parameterised transition);
$\vartheta \in (\mathcal{V}_S \to U)$ a valuation for the location variables of l;
$\varphi \in T_\Sigma(\mathsf{Var}_S)$ a constraint (switch restriction of incoming transition of l in S);
$\rho \in (\mathcal{V}_S \to T_\Sigma(\mathsf{Var}_S))$ update mapping of incoming transitions of l in S.
Note that we include information in APTS states (i) for subsequent use, and (ii) to avoid unintended overlap between the intermediate parameterised APTS states which we construct as discussed in case 2. in the next paragraph.

APTS transitions We map STS transitions $l \xrightarrow{g,\varphi,\rho} l'$ onto transitions in the APTS as follows:

1. We map each STS transition that has an empty list of interaction variables, i.e. where $\mathsf{arity}(g) = 0$, onto a single (instantiated) transition in the APTS.
2. We map each other STS transition, i.e. where $\mathsf{arity}(g) > 0$, onto the combination of
   - a parameterised transition t,
   - an intermediate parameterised state s'' reached by t, and
   - a set of instantiations i that link s'' with an instantiated state s'.

Location variables and update mapping For each switch t_S in the STS, and for each corresponding APTS transition t_P, the set of interaction variables is fully determined by the gate g of t_S resp. t_P. In the APTS, the source state of any transition t_P has an empty instantiation obligation, and thus, the instantiation obligation of the APTS state reached by t_P consists of (location variables corresponding to) the interaction variables of t_P. In the APTS, for each interaction variable $x \in \mathcal{I}_S = \mathcal{I}_P$, we always use the same location variable x_L to represent x in the instantiation obligation. Thus, for each gate $g \in \mathcal{G}_P$ we always use the same set of location variables $\mathcal{V}_g$ to represent the interaction variables $\mathsf{params}(g)$, and an update mapping ρ_g such that $\mathcal{V}_g$ is bijective with $\mathsf{params}(g)$, as made explicit by bijective function $\rho_g : \mathcal{V}_g \to \mathsf{params}(g)$. We use $\mathcal{V}_g$ and ρ_g in the SOS rules in Definition 6.4.1.

Definition 6.4.1
Let $\langle S, \iota \rangle$ be an initialised STS with STS $S = \langle L_S, l_{0,S}, \mathcal{V}_{L,S}, \mathcal{I}_S, \mathcal{G}_S, \to_S \rangle$, and initialisation $\iota \in (\mathcal{V}_{L,S}(l_0) \to U)$, where
$\mathcal{V}_S \;=_{\text{def}} \bigcup_{l \in L_S} \mathcal{V}_{L,S}(l)$, and
$\mathsf{Var}_S \;=_{\text{def}} \mathcal{V}_S \cup \mathcal{I}_S$.
The corresponding APTS P is defined as $P = \langle L_P, l_{0,P}, \mathcal{V}_{L,P}, \mathcal{I}_P, \mathcal{G}_P, \to_P, \Phi, \cdots \rangle$, where

$L_P = L_S \times (\{\emptyset\} \cup \{\mathcal{V}_g \mid g \in \mathcal{G}_S\}) \times (\mathcal{V}_S \to U) \times T_\Sigma(\mathsf{Var}_S) \times (\mathcal{V}_S \to T_\Sigma(\mathsf{Var}_S))$
is the set of states,
$l_{0,P} = \langle l_{0,S}, \emptyset, \iota, \mathsf{true}, \emptyset \rangle$ is the initial state,
$\mathcal{V}_{L,P}$ is projected on the second component of a state,
$\mathcal{I}_P = \mathcal{I}_S$,
$G_P = G_S$, and
Φ is projected on the fourth component of a state,
$\to_P$ and $\cdots\!\!\succ$ are defined using SOS rules below.

We map STS transitions as discussed above:

1. for the case where $\mathsf{arity}(g) = 0$ the APTS contains a single instantiated transition (solid arrow):

$$\text{pf:}\ \frac{l \xrightarrow{g,\varphi,\rho} l' \qquad \mathsf{arity}(g) = 0 \qquad \vartheta \models \varphi \qquad \vartheta' = \vartheta_{\mathsf{eval}} \circ \rho}{\langle l, \emptyset, \vartheta, \mathsf{true}, \emptyset \rangle \xrightarrow{g,\mathsf{true},\rho_g} \langle l', \emptyset, \vartheta', \mathsf{true}, \emptyset \rangle}$$

2. for the case where $\mathsf{arity}(g) > 0$ the APTS contains

   (rule pm-t) a single parameterised transition t (dashed arrow), with

   (rule pm-i) for each possible valuation of the parameters of t that satisfies the constraint, an instantiation (dotted arrow).

   The parameterised label contains parameters, and a constraint which is obtained from the switch restriction. Note that we assume that we can represent the switch restriction φ, updated with the valuation ϑ, as a term $\in T_\Sigma^{\mathsf{bool}}(\mathsf{params}(g))$.

$$\text{pm-t:}\ \frac{\begin{array}{c} l \xrightarrow{g,\varphi,\rho} l' \qquad \mathsf{arity}(g) > 0 \qquad \varsigma \in (\mathsf{params}(g) \to U) \\ \vartheta \cup \varsigma \models \varphi \qquad\qquad \vartheta' = (\vartheta \cup \varsigma)_{\mathsf{eval}} \circ \rho \end{array}}{\langle l, \emptyset, \vartheta, \mathsf{true}, \emptyset \rangle \dashrightarrow \langle l', \mathsf{params}(g), \vartheta, \varphi[\vartheta], \rho \rangle}$$

$$\text{pm-i:}\ \frac{\text{same premiss as in pm-t}}{\langle l', \mathsf{params}(g), \vartheta, \varphi[\vartheta], \rho \rangle \cdots\!\!\overset{\varsigma \circ \rho_g}{\cdots}\!\!\succ \langle l', \emptyset, \vartheta', \mathsf{true}, \emptyset \rangle}$$

□

Correctness It is in principle possible to prove that the LTS that can be obtained directly from an instantiated STS $\langle S, \iota \rangle$ by using Definition 6.1.2, is bisimilar with the LTS that can be obtained by first mapping the STS on an APTS, using Definition 6.4.1, and then obtaining the LTS from the APTS, using Definition 6.3.2. We do not give the proof here.

Implementation note In the implementation of the Explorer component that implements the APTS access to an STS model, we used the following optimisation. We use information from the switch restrictions of the STS to obtain bindings for interaction variables of which the value is fully determined by the switch restriction, such that we can create transitions where some, or all, of the parameters are instantiated. For example, as shown in Figure 6.6, we create a transition $l \xrightarrow{!\mathsf{out}(0)} l'$, instead of a transition $l \xrightarrow{!\mathsf{out}(x:nat)[x=0]} l''$ with an instantiation $l'' \cdots\!\!\overset{x \leftarrow 0}{\cdots}\!\!\succ l'$.

6.4.2 Example: Music Player

We now show the construction of the APTS for the music player of Section 6.2.1. This APTS is infinite, because parameter (interaction variable) n of $?\mathsf{init}(n:\mathsf{nat})$ has an infinite domain: it ranges over all natural numbers. In Figure 6.7 we thus show only part of this APTS. In the discussion below, we look at states of the APTS, and for each of the states that we discuss, we look at the outgoing transitions.

Initial state The initial state of the APTS is $\langle i, \emptyset, \emptyset, \mathsf{true}, \emptyset\rangle$. It contains (1) the initial location of the STS (i), (2) the set of interaction variables that have to be instantiated ($\emptyset$), (3) the valuation of the variables of location i ($\emptyset$), (4) the constraint that must hold over the variables (true), and (5) the update mapping ($\emptyset$). The set of interaction variables that have to be instantiated, and the update mapping are empty, because we are in the initial location of the STS. For the same reason, the constraint is true.

From initial location i of the STS there is a single outgoing switch:

$$i \xrightarrow{?\mathsf{init},\mathsf{true},nsong:=n} s$$

This switch is enabled, because its switch restriction is true. Thus, in the APTS there is a parameterised transition, t_0, from the initial state. The destination state of t_0 differs in two aspects from its source state: (1) it contains location s (the destination of the switch in the STS), and (2) it contains parameter n in the set of free variables, because n is introduced in the label on transition t_0.

$$\langle i, \emptyset, \emptyset, \mathsf{true}, \emptyset\rangle \dashrightarrow^{?\mathsf{init}(n\,:\,\mathsf{nat})} \langle s, \{n\}, \emptyset, \mathsf{true}, \emptyset\rangle \qquad (t_0)$$

State $\langle s, \emptyset, \mathsf{true}, \emptyset\rangle$ From the destination of transition t_0 there is an infinite number of instantiations, of the form shown below as $i_{0,\mathtt{j}}$, that assign value $\mathtt{j}$ to n, where we use $\mathtt{j}$ as a named constant, representing any natural number. The source state of each such instantiation is the destination of transition t_0; the destination state of the instantiation is obtained by updating the source state to show the effect of the instantiation:

1. parameter n is removed from the set of free variables, and
2. variable *nsong* is set to $\mathtt{j}$.

$$\langle s, \{n\}, \emptyset, \mathsf{true}, \emptyset\rangle \cdots^{n=\mathtt{j}}\cdots\!\!\succ \langle s, \emptyset, \{nsong{\leftarrow}\mathtt{j}\}, \mathsf{true}, \emptyset\rangle \qquad (i_{0,\mathtt{j}})$$

Of the infinite number of concrete instances of these instantiation transitions, in Figure 6.7 we show only the following ones:

$$\langle s, \{n\}, \emptyset, \mathsf{true}, \emptyset\rangle \cdots^{n=0}\cdots\!\!\succ \langle s, \emptyset, \{nsong{\leftarrow}0\}, \mathsf{true}, \emptyset\rangle \qquad (i_{0,0})$$

$$\langle s, \{n\}, \emptyset, \mathsf{true}, \emptyset\rangle \cdots^{n=1}\cdots\!\!\succ \langle s, \emptyset, \{nsong{\leftarrow}1\}, \mathsf{true}, \emptyset\rangle \qquad (i_{0,1})$$

$$\langle s, \{n\}, \emptyset, \mathsf{true}, \emptyset\rangle \cdots^{n=2}\cdots\!\!\succ \langle s, \emptyset, \{nsong{\leftarrow}2\}, \mathsf{true}, \emptyset\rangle \qquad (i_{0,2})$$

$$\langle s, \{n\}, \emptyset, \mathsf{true}, \emptyset\rangle \cdots^{n=3}\cdots\!\!\succ \langle s, \emptyset, \{nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \qquad (i_{0,3})$$

Now, we look at the transitions that are enabled from the destinations of these instantiations.

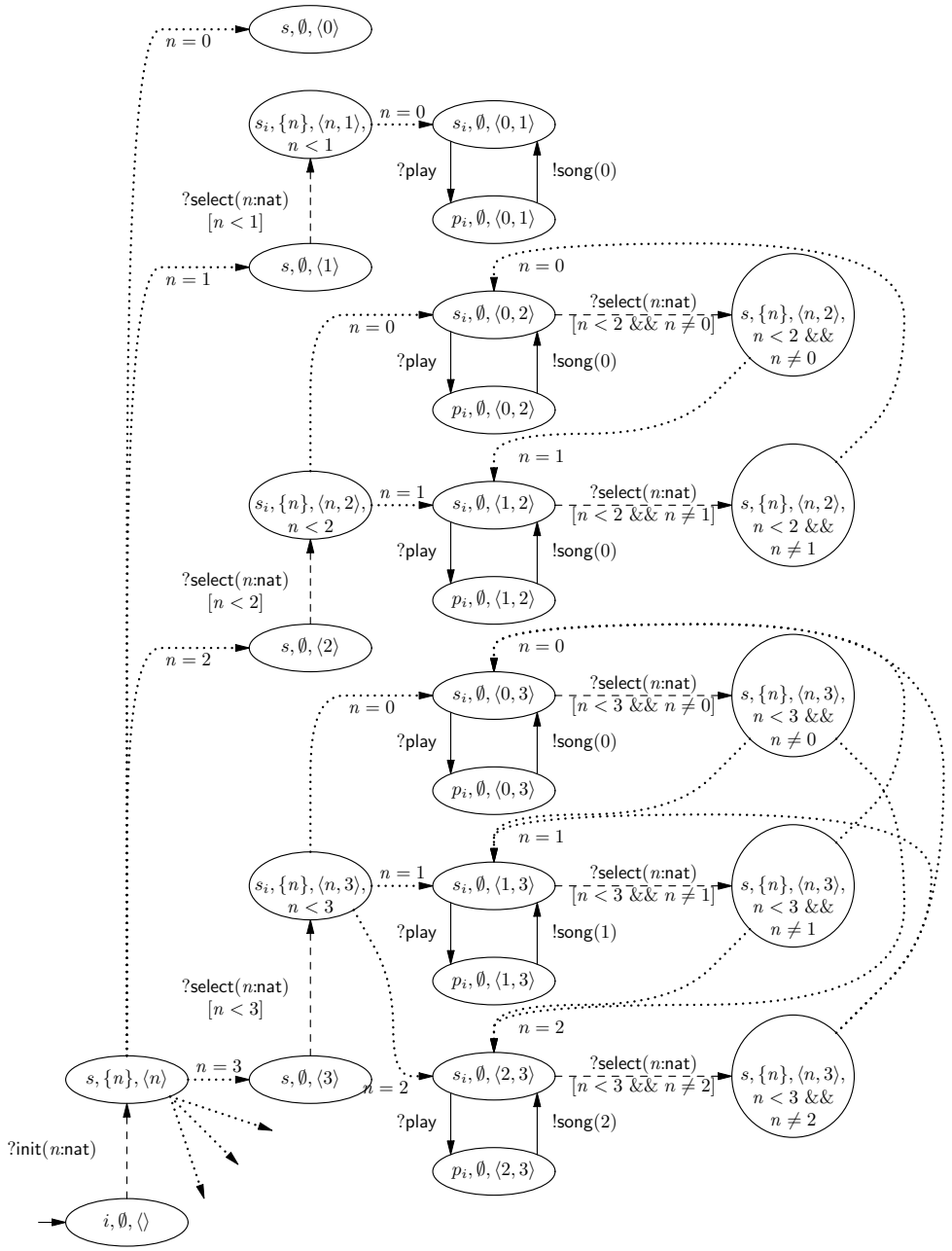


Figure 6.7: Partial APTS of the music player—showing full behaviour for 0, 1, 2 and 3 songs. States show, from left to right: STS location, free variables, the values of *cur* and *nsong*, when relevant for the given state, and (when applicable) the switch restriction in which values of *cur* and *nsong* have been instantiated.

States $\langle s, \emptyset, \{nsong \leftarrow \mathsf{j}\}, \mathsf{true}, \emptyset\rangle$ Each of these destinations corresponds to location s in the STS. This STS location has only a single outgoing switch:

$$s \xrightarrow{?\mathsf{select}, n<nsong, cur:=n} s_i$$

The switch has label ?select and switch restriction $n < nsong$. For the destination of instantiation $i_{0,0}$ it is the case that $nsong = 0$; when we substitute this value in the switch restriction we get $n < 0$, and there are no natural numbers n for which this switch restriction holds. Therefore, the destination of $i_{0,0}$ has no outgoing transitions in the APTS. For each of the other destinations, it is the case that $nsong >= 1$, and thus, for those destinations, there is at least one natural number n for which the switch restriction $n < nsong$ holds. In Figure 6.7 we show the outgoing transitions for the destinations of instantiations $i_{0,0}$, $i_{0,1}$, $i_{0,2}$ and $i_{0,3}$; we only discuss the outgoing transitions of the destination of instantiation $i_{0,3}$.

State $\langle s, \emptyset, \{nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle$ Now, we look at the transitions that are enabled from the destination of instantiation $i_{0,3}$. As we have seen above, from location s in the STS there is only one outgoing switch, and, as we discussed, there is at least one solution for its switch restriction. Thus, there is a corresponding transition in the APTS:

$$\begin{aligned}\langle s, \emptyset, \{nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle &\dashrightarrow^{?\mathsf{select}(n:\mathsf{nat})[n<3]} \\ &\langle s_i, \{n\}, \{nsong{\leftarrow}3\}, n < 3, \emptyset\rangle\end{aligned} \quad (t_1)$$

State $\langle s_i, \{n\}, \{nsong \leftarrow 3\}, n < 3, \emptyset\rangle$ From the destination of transition t_1 there is a finite number of instantiation transitions, one for each natural number $n < 3$:

$$\langle s_i, \{n\}, \{nsong{\leftarrow}3\}, n < 3, \emptyset\rangle \cdots^{n=0}\rightarrow \langle s_i, \emptyset, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \quad (i_{1,0})$$

$$\langle s_i, \{n\}, \{nsong{\leftarrow}3\}, n < 3, \emptyset\rangle \cdots^{n=1}\rightarrow \langle s_i, \emptyset, \{cur{\leftarrow}1, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \quad (i_{1,1})$$

$$\langle s_i, \{n\}, \{nsong{\leftarrow}3\}, n < 3, \emptyset\rangle \cdots^{n=2}\rightarrow \langle s_i, \emptyset, \{cur{\leftarrow}2, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \quad (i_{1,2})$$

For each of these instantiations, there are two outgoing transitions from their destination, one parameterised one, and one instantiated one.

State $\langle s_i, \emptyset, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle$ From the destination of instantiation $i_{1,0}$ these two transitions are:

$$\begin{aligned}\langle s_i, \emptyset, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle &\dashrightarrow^{?\mathsf{select}(n:\mathsf{nat})[n<3 \;\&\&\; n\neq 0]} \\ &\langle s_i, \{n\}, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, n < 3 \;\&\&\; n \neq 0, \emptyset\rangle\end{aligned} \quad (t_2)$$

$$\langle s_i, \emptyset, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \xrightarrow{?\mathsf{play}} \langle p_i, \emptyset, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \quad (t_3)$$

State $\langle s_i, \{n\}, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, n < 3 \;\&\&\; n \neq 0, \emptyset\rangle$ From the destination of transition t_2 there are two instantiations; both bring us to states that we have visited before—the destinations of instantiations $i_{1,1}$ resp. $i_{1,2}$:

$$\langle s_i, \{n\}, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, n < 3 \;\&\&\; n \neq 0, \emptyset\rangle \xrightarrow{n=1} \langle s_i, \emptyset, \{cur{\leftarrow}1, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \tag{$i_{2,0}$}$$

$$\langle s_i, \{n\}, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, n < 3 \;\&\&\; n \neq 0, \emptyset\rangle \xrightarrow{n=2} \langle s_i, \emptyset, \{cur{\leftarrow}2, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \tag{$i_{2,1}$}$$

State $\langle p_i, \emptyset, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle$ From the destination of transition t_3 there is a single transition that brings us back to the destination of instantiation $i_{1,0}$

$$\langle p_i, \emptyset, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \xrightarrow{!\mathsf{song}(0)} \langle s_i, \emptyset, \{cur{\leftarrow}0, nsong{\leftarrow}3\}, \mathsf{true}, \emptyset\rangle \tag{t_4}$$

With this transition we show the effect of the optimisation mentioned in the implementation note at the end of Section 6.4. The STS contains switch $p_i \xrightarrow{!\mathsf{song}, n=cur, \emptyset} s_i$, where $\mathsf{params}(!\mathsf{song}) = (n)$. Effectively, the switch restriction $n = cur$ equates the interaction variable n with location variable cur. The value of cur is given in the valuation: $cur{\leftarrow}0$. Combined, we get the APTS transition with label $!\mathsf{song}(0)$.

6.5 Testing with an alternating PTS

In the remainder of this chapter, when we discuss testing based on PTSes, we will restrict ourselves to well-defined alternating PTSes. Before we discuss how we extend our architecture to support on-line testing with APTSs, we first discuss how testing with an APTS differs from testing with an LTS.

A first difference is with the labels: we use parameterised and instantiated labels, instead of "plain" LTS labels. For the other differences, we structure the discussion by the main activities in our approach to on-line testing:

1. computing the potential behaviour for the next step;
2. interacting with the SUT;
3. updating the tester state and giving a verdict, if one is available.

Only the first and last activity are affected by the use of an APTS instead of an LTS. The second activity (interaction with the SUT) is (by design) unaffected.

Labels

In the previous chapters we used opaque labels, introduced as part of the LTS definition. Now we decompose these labels. Two kinds of labels are exchanged over our interfaces, as we explain using Fig. 6.8.

1. instantiated labels are exchanged in the Test Execution part of our architecture, i.e. between Manager and Adapter;
2. parameterised labels are exchanged in the Test Derivation part.

Ad 1: instantiated labels An instantiated label is a tuple $\langle g, \overline{t} \rangle$ where g is a gate, and each t_i in $\overline{t}$ is a ground term. Each such ground term is the canonical term representation of a value in U. Examples of such labels are $!\mathsf{out}(0)$ and $!\mathsf{song}(0)$.

Ad 2: parameterised labels A parameterised label is a tuple $\langle g, \overline{x}, \varphi \rangle$ where g is a gate, each x_i in $\overline{x}$ is a variable $\in \mathsf{params}(g)$, and φ is a constraint $\in T_{\Sigma}^{\mathsf{bool}}(\overline{x})$ over the variables in $\overline{x}$. Examples of such labels are $?\mathsf{in}(x : \mathsf{nat})$ and $?\mathsf{select}(n : \mathsf{nat})[n < 3]$.

We use the following additional notation.

- we will use subscripts i and p, as in e.g. L_i resp. L_p, to explicitly indicate whether instantiated or parameterised labels are exchanged.
- $\mathsf{act}(l)$ denotes the gate of a label,act i.e. $\mathsf{act}(\langle g, \overline{x}, \varphi \rangle) =_{\text{def}} g$.
- $\mathsf{term}(l)$ denotes the sequence of terms or variables of a label, i.e. $\mathsf{term}(\langle g, \overline{t} \rangle) =_{\text{def}} \overline{t}$ andterm $\mathsf{term}(\langle g, \overline{x}, \varphi \rangle) =_{\text{def}} \overline{x}$.
- $\mathsf{constraint}(l)$ denotes the constraint in a label, i.e. $\mathsf{constraint}(\langle g, \overline{t} \rangle) =_{\text{def}} \mathsf{true}$ and$\mathsf{constraint}$ $\mathsf{constraint}(\langle g, \overline{x}, \varphi \rangle) =_{\text{def}} \varphi$.
- $\mathsf{var}(l)$ denotes the variables in a label, i.e. $\mathsf{var}(\langle g, \overline{t} \rangle) =_{\text{def}} ()$ andvar $\mathsf{var}(\langle g, \overline{x}, \varphi \rangle) =_{\text{def}} \overline{x}$.

Representation of quiescence To denote quiescence as parameterised (instantiated) label, we use $\langle \delta, (), \mathbf{true} \rangle$, a parameterised label with action δ, an empty sequence of terms $()$, and constraint **true**; we will typically abbreviate this to δ.

We continue to use L^{δ} to denote the label set L, extended with quiescence.

6.5.1 Computation of potential behaviour

We compute the potential behaviour, using the enabled outgoing transitions of our current states in the APTS. The computation of the potential behaviour involves the following activities:

1. obtain the enabled outgoing transitions of our current states in the APTS;
2. compute the possible stimuli and the expected responses;
3. compute which of the current APTS states are quiescent, to be able to synthesise δ-labelled transitions;
4. combine the labels of the enabled transitions such that (a) we can easily select a stimulus, when we decide to apply a stimulus as next test step, and (b) we can easily update our state, after the interaction with the SUT has taken place. This corresponds to the on-the-fly determinization that we do in the LTS setting.

Ad 1: obtaining the enabled transitions Obtaining the enabled outgoing APTS transitions from a given model is the task of the modelling-language-specific Explorer component for the respective modelling language. According to the SOS rules that describe the derivation of an APTS from an STS, we only add

a transition to the APTS when the constraint on the corresponding STS switch is satisfiable, i.e. when a valuation of the variables in the constraint exists, such that evaluating it yields true. In general, satisfiability checking is undecidable, i.e. the result of such computation may be either of three possibilities: (1) no valuation exists, (2) a valuation exists, or (3) it is unknown whether a valuation exists.

Thus, there may be switches in the STS model, for which an Explorer cannot decide whether or not they are enabled. The source of the undecidability lies in the presence of variables (interaction variables in the STS) in the guard (switch restriction). We assume that, once a valuation is known, such a constraint can simply be evaluated.

Our approach to deal with this problem of undecidability is twofold. Firstly, we assume that in most cases data types and constraints (switch restrictions) in any practical STS model are chosen such that satisfiability of the constraints is decidable. Secondly, in the implementation of our test tool design we provide a "safety net" for the (rare, we assume) case that our decidability assumption does not hold.

For this "safety net" we use the ability of the Explorer to report its inability to decide on satisfiability of an constraint, when we add transitions to the APTS:

- If no valuation exists, the respective transition is excluded from the APTS.
- If a valuation exists, the respective transition is included in the APTS.
- When it is unknown whether a valuation exists, the respective transition *is* included in the APTS, but with a special mark that indicates its special status, and this mark plays a role in the subsequent activities.

Ad 2: computation of possible stimuli and expected responses We obtain the sets of possible stimuli and expected responses from the set of enabled APTS transitions, using the IO-Oracle. We include the transitions for which enabledness could not be decided. However, we mark stimuli obtained from transitions for which enabledness could not be decided, so the random testing strategy can avoid them when choosing a stimulus (if we are unable to decide satisfiability, it is likely that we are also be unable to come up with an instantiation for the label parameters). When we, in this way, skip a potential stimulus while deciding on the next test step, we add an entry into the test run log, to allow subsequent analysis of omitted stimuli.

Ad 3: computation of quiescent states In the LTS setting, we mark states as quiescent when they do not have any outgoing internal or output transitions (or when they are divergent, when the corresponding setting is enabled).

In the APTS we may have states

- without outgoing output or internal transitions. These states are quiescent, just like in the LTS setting.
- with at least one outgoing output or internal transition of which the enabledness could be decided. These states are not quiescent, just like in the LTS setting.
- that have one or more outgoing output or internal transitions, all of which have the special mark, i.e. enabledness could not be decided. These states

are *conditionally quiescent*, which we explain below.
A conditionally quiescent state is quiescent when, for each of the outgoing output transitions, the constraint does not hold. The corresponding condition (the conjunction of, for each of the outgoing output transitions, the negation of the constraint) is associated with the state. We return to this when we discuss how we update the tester state and give verdicts, below.

Ad 4: combining labels In the LTS setting we perform on-the-fly determinization in the *unfold* function in the Primer: during the computation of the potential behaviour we collect sets of enabled input and output labels, and with each observable label l in these sets, we associate a set containing the states reached by traversing a transition with l. Thus, when, after an interaction with the SUT, represented by label l, we can immediately "take" the set of states associated with l, to get the set of states, reached by model transitions with l.

In the APTS setting we can not determinize in this way. We may have parameterised labels of which the constraints "overlap", and then a single instantiated label may be "an instance of" multiple, distinct, parameterised labels. For example, if we have parameterised labels $?\mathsf{in}(x : \mathsf{nat})[x < 3]$ and $?\mathsf{in}(x : \mathsf{nat})[x > 1 \wedge x < 4]$, then an instantiated label $?\mathsf{in}(2)$ will match with both. To a certain extent we can use the gates of the labels to distinguish between them, but given parameterised and instantiated labels with the same gate, we cannot immediately say whether two distinct parameterised labels have the same, or overlapping, or distinct instantiations, nor can we immediately say whether a given instantiated label is an instance of the parameterised labels. To check whether an instantiated label is an instance of parameterised ones, we have to look in the APTS, and see whether the states that are reached by the parameterised labels have an instantiation that corresponds to the instantiated label.

In the APTS setting we perform so-called *pre-determinization*. We collect sets of parameterised labels l_p, and with each parameterised label in these sets, we associate a set containing the states reached by traversing a transition with l_p. When we have two parameterised labels $\langle g, \overline{x}, \varphi_1 \rangle$ and $\langle g, \overline{x}, \varphi_2 \rangle$, that only differ in their constraints (φ_1 vs. φ_2), we will use them as two distinct items in the sets of possible stimuli and expected responses. We do this, to expose the control structure of the model in the potential behaviour. Thus, we will treat $?\mathsf{in}(x : \mathsf{nat})[x < 3]$ and $?\mathsf{in}(x : \mathsf{nat})[x \neq 2 \text{ and } x < 3]$ as distinct items. Given an instantiated label l_i (representing interaction with the SUT), there may be *multiple* parameterised labels that match l_i. For example, given $l_i = ?\mathsf{in}(1)$, both parameterised labels that we gave above match. Thus, for both labels we will have to check whether the APTS states, reached by transitions with these labels, have instantiations $x \leftarrow 1$, to compute the successor behaviour.

6.5.2 Interaction with the SUT

Interaction with the SUT is in the APTS setting the same as in the LTS setting, except that interactions (with the SUT) are now represented by instantiated labels.

6.5.3 Updating the tester state

After the interaction with the SUT has taken place, we have to update the tester state, and give a verdict, if one is available. We discuss two cases:

1. interaction took place (stimulus was applied, or response obtained), and
2. no interaction took place (quiescence was observed).

Ad 1: interaction In this case, we have an instantiated label l_i that represents the interaction with the SUT. As we have discussed above, there may be more than one parameterized label l_p of which l_i is an instance, and thus, we have to check all candidates, i.e. all parameterized labels that have the same gate as the instantiated label. With each such candidate label l_i we have the set of APTS states reached by a transition with l_i; for each of these states we check whether it has an outgoing instantiation that corresponds to l_i. Even when for some of the labels, satisfiability of the constraints could not be decided, an instantiated label gives us a valuation for the parameters of such constraint, such that it can simply be evaluated.

If none of the candidates have an instantiation that corresponds to i_l, the test run will end, and a verdict will be given: a **fail** verdict if l_i is an output (response), and an error verdict otherwise—any stimulus that we apply should be present in the model.

Ad 2: quiescence In this case, we check whether quiescence was expected—if not, we issue a **fail** verdict. If quiescence was expected, we update the tester state to the set of quiescent states (independent of whether they were conditionally quiescent). In addition, we check whether we have conditionally quiescent states. If so, we add their associated conditions to the test run log, and we add a note that tells whether all quiescent states (of the current tester state) are conditionally quiescent. At the end of the test run, the conditions collected in the test run log form a proof obligation: the test run verdict is only valid when the conditions hold.

6.6 Extension of Architecture

We now look at the changes necessary to allow the use of an alternating PTS as underlying formalism in our architecture, where we restrict ourselves to random on-line testing.

This section is structured as follows. We first look at the extension of our architecture with additional components (Section 6.6.1), and discuss new and changed interfaces (Section 6.6.2); then we discuss changes to the algorithms of Primer (Section 6.6.3) and Manager (Section 6.6.4).

6.6.1 Components

As indicated in the introduction to this chapter, we use the PTS formalism to have a more concise (finite) representation of potential behaviour (possible stimuli and expected responses), using parameterised labels. However, we do

not let this affect the interface between Manager and Adapter: there we still exchange LTS labels (to which we now refer as instantiated labels). Thus, we have introduced a "gap" between the labels that the Manager gets from the DerivationEngine, and the labels that the Manager has to pass to the Adapter when it wants to apply a stimulus. For example, in the case of the music player, the potential behaviour of the initial state consists of the parameterised label ?init(n:nat). The Manager cannot pass that label to an Adapter: an Adapter needs an instantiated label, like ?init(3).

The role of the Instantiator, the sole component that we add to the architecture (see Fig. 6.8), is to bridge this gap: the Manager can ask the Instantiator to instantiate a parameterised label. The Instantiator may need information from the model to do its work, hence the dotted arrow from Specification to Instantiator in the figure.

(The labels that the Manager gets from the Adapter, as representation of observations, are instantiated labels. As we will see, the Manager can, as before, pass such label to the DerivationEngine in interface function next).

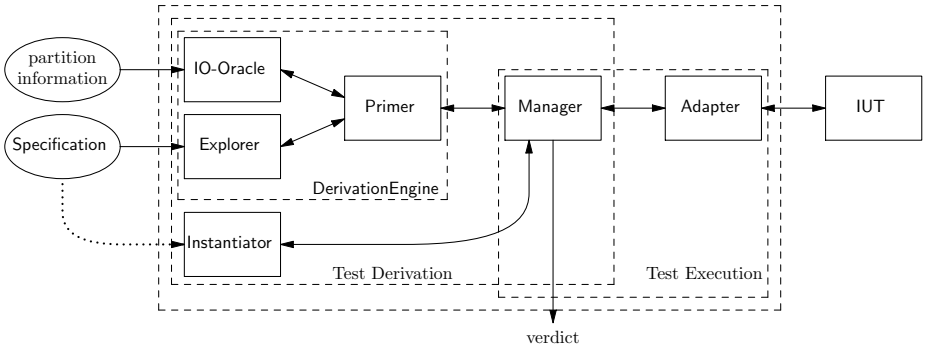


Figure 6.8: Our architecture for on-line model-based testing of Fig. 3.2 and Fig. 4.4, extended with Instantiator

6.6.2 Interfaces

An immediate consequence of the change from an LTS-based DerivationEngine interface to an APTS-based one, is that we refine (decompose) the labels. We discuss this first. Then we discuss the new Instantiator interface, and we discuss the impact of the change to APTS-based interfaces.

Interfaces

Since our aim is to change from an LTS-based DerivationEngine interface to a PTS-based one, obviously we have to update the DerivationEngine (i.e. Primer) interface, and we also have to update the Explorer interface. The interface between the IO-Oracle and the Primer is not changed (except that Primer now passes parameterised labels over the interface); we do not discuss it further. We start by discussing the interface between Instantiator and Manager.

Between Instantiator and Manager: the Instantiator interface

The Instantiator must implement one interface function, of which we give first the signature (in Table 6.1), and then the definitions, both in terms of a PTS $\langle S, s_0, \mathcal{V}_L, \mathcal{I}, \mathcal{G}, \rightarrow, \Phi, \cdots \rangle$[2]:

1. getInstance : $L_p \rightarrow L_i \uplus \{\bot\}$

Table 6.1: Signature of Instantiator interface functions.

Ad 1: getInstance Function getInstance must instantiate the parameters in a given parameterised label, satisfying the constraint, and return the resulting instantiated label. We describe its functionality using instances(l).

$$\mathsf{instances}(l) \ =_{\text{def}} \ \{l[\varsigma] \mid \varsigma \models \mathsf{constraint}(l) \text{ and } l[\varsigma] \text{ is correctly typed}\} \quad (6.1)$$

Function getInstance(l) returns an element from instances(l) when instances(l) $\neq \emptyset$, or $\bot$ otherwise. The choice, which element to return (when instances(l) when instances(l) $\neq \emptyset$), is left to the implementation.

Between Explorer and Primer: The Explorer interface

In Chapter 4 we defined the Explorer interface that provides access to an LTS in Table 4.2. We now extend this to allow the Explorer to provide access to a PTS $\langle S, s_0, \mathcal{V}_L, \mathcal{I}, \mathcal{G}, \rightarrow, \Phi, \cdots \rangle$, as follows:

1. start $:\rightarrow S$
2. menu : $S \rightarrow \mathcal{P}(L_{p,\tau} \times (\mathcal{V} \rightharpoonup \mathcal{I}) \times S) \uplus \{\bot\}$
3. inst : $S \times (\mathsf{Vars} \rightharpoonup T_\Sigma) \rightarrow S \uplus \{\bot\}$

Table 6.2: Signature of PTS-based Explorer interface functions.

Ad 1: start Function start is not changed, i.e. its definition remains:

$$\mathsf{start}() \ =_{\text{def}} \ s_0 \quad (6.2)$$

Ad 2: menu Function menu now returns parameterised labels, and a mapping between the parameters (interaction variables) in the labels and the associated set of location variables—we need this mapping to generate correct bindings for the inst function. Moreover, it can return an error (represented by $\bot$) when the interface user asks for the menu of a parameterised state (whether it returns an

[2]We use S and s_0 for the set of states resp. the initial state, to be able to use L, L_i, L_p etc. for labels.

error, or just returns a menu, depends on whether the Explorer allows delayed instantiation). It is defined as follows:

$$\mathsf{menu}(p) \;=_{\text{def}}\; \begin{cases} \{\langle\langle g, \mathsf{params}(g), \varphi\rangle, \rho, s'\rangle \mid \exists p \xrightarrow{g,\varphi,\rho} s' \in \rightarrow\} & \text{if } \mathcal{V}_L(p) = \emptyset \\ \{\bot\} & \text{otherwise} \end{cases} \tag{6.3}$$

Ad 3: inst The new function inst is given a parameterised state p and a set of bindings b (here represented as a partial mapping from variables to terms). If the PTS contains a corresponding instantiation i (of which the set of bindings b' is contained in b—bindings $b \setminus b'$ are simply ignored), function inst returns the destination state of i; otherwise it returns an error (represented by $\bot$). It is defined as follows:

$$\mathsf{inst}(p, b) \;=_{\text{def}}\; \begin{cases} s' & \text{if } \exists p \cdots\!\overset{b'}{\cdots}\!\cdots\!\triangleright\, s' \in \cdots\!\triangleright \text{ such that } b' \subseteq b \\ \{\bot\} & \text{otherwise} \end{cases} \tag{6.4}$$

Provided by the Primer: The DerivationEngine interface

In Chapter 3 we defined the LTS-based interface to the DerivationEngine by giving the signature (see Table 3.8), and the definition (see Table 3.9 for the unguided case) of the interface functions. In Chapter 4 we showed how the Primer component implements this interface. To be able to interact with the parameterized transition system, we extend this interface to the following (the interface contains the same functions, only the signatures of the functions that accept or return labels have changed):

1. start $:\rightarrow P \times (L_V \uplus \{\bot\})$
2. in $: P \rightarrow \mathcal{P}(L_{I,p})$
3. hasOutputs $: P \rightarrow \mathit{bool}$
4. out $: P \rightarrow \mathcal{P}(L^{\delta}_{U,p} \times (L_V \uplus \{\bot\}))$
5. next $: P \times L^{\delta}_i \rightarrow (P \uplus \{\bot\}) \times (L_V \uplus \{\bot\})$
6. defNegVerdict $:\rightarrow L_V$
7. defPosVerdict $:\rightarrow L_V$

Table 6.3: Signature of PTS-based Primer functions. Type P is the pseudo-state type.

Ad 1, 3, 6, 7: The signatures of functions start, hasOutputs, defNegVerdict and defPosVerdict are not changed.

Ad 2: in Function in returns parameterized labels.

Ad 4: out Function out returns parameterized labels.

Ad 5: next Function next takes an instantiated label as its second argument.

6.6.3 Extension of Primer algorithm

We now discuss the changes to the Primer. Our choice for a concise, symbolic, representation of the potential behaviour, affects how we deal with determinization, and with actual behaviour.

We first discuss the changes to the Primer pseudo-state type, and then we discuss changes to the DerivationEngine interface implementation.

The specification-only DerivationEngine pseudo-state type

In Definition 4.2.1 we defined the pseudo-state type for the DerivationEngine for random testing as a tuple $\langle m_d, \mathit{unfoldResult}\rangle$, with *unfoldResult* a tuple $\langle i, o, n_l, m_e\rangle$, and in Table 4.4 we showed the methods implemented on it.
We change the following fields of pseudo-state type element *unfoldResult*:

1. field i, to have signature $\mathcal{P}(L_I \times (\mathcal{V} \rightharpoonup \mathcal{I}))$,
2. field o, to have signature $\mathcal{P}((L_U^{\delta}) \times (L_V \uplus \{\bot\}) \times (\mathcal{V} \rightharpoonup \mathcal{I}))$, and
3. field n_l, to have signature $\mathit{Gates} \to ((T_\Sigma(\mathsf{Var})^* \times T_\Sigma^{\mathsf{bool}}(\mathsf{Var})) \to \mathcal{S})$.

Moreover, we change two methods:

4. method *unfold*, and
5. method n (used by interface function next).

Ad 1: field i Whereas, in the non-symbolic case, field i contains a set of (input) labels, it now contains a set of tuples, where each tuple consists of a parameterized label and a mapping between location and interaction variables.

Ad 2: field o Whereas, in the non-symbolic case, field o contains a set of tuples, where each tuple consists of a label and a verdict (or $\bot$), now the tuples contain one additional element: a mapping between location and interaction variables.

Ad 3: field n_l Whereas, in the non-symbolic case, field n_l contains a mapping from labels to sets of states, it now contains a two-level mapping. Given a label l, the first-level mapping maps label gates ($\mathsf{act}(l)$) to the second-level mapping, which maps variables-constraint tuples ($\langle \mathsf{term}(l), \mathsf{constraint}(l)\rangle$) to a set of states. Thus, the type of the mapping is:

$$n_l : \mathit{Gates} \to ((T_\Sigma(\mathsf{Var})^* \times T_\Sigma^{\mathsf{bool}}(\mathsf{Var})) \to \mathcal{S})$$

Ad 4: method *unfold* In Algorithm 6.1 we show the modifications to method unfold. The changes are the following:

- in line 4, the initialization of *destmap* is changed to reflect the two-level mapping of field n_l;

Algorithm 6.1: *P.unfold*() for PTS Primer— changes w.r.t. Algo 4.1

```
   input : P, a pseudo-state
           e, an Explorer that gives access to the Specification
           o, an IO-Oracle
   output: P, the pseudo-state, unfolded if it wasn't already unfolded
 1 begin
 2   if P.unfoldResult = ⊥ then
 3     ⋮
 4     destmap ⟵ {g → (⟨t̄, φ⟩ → ∅) | ⟨g, t̄, φ⟩ ∈ L^δ}
 5     while work ≠ ∅ do
 6       ⋮
 7       foreach ⟨l = ⟨g, x̄, φ⟩, ρ, p'⟩ ∈ e.menu(p) do
 8         kind ⟵ o.kind(l)
 9         if kind = I then
10           inputs ⟵ inputs ∪ {l}
11           destmap[g][⟨x̄, φ⟩] ⟵ destmap[g][⟨x̄, φ⟩] ∪ {⟨ρ, p'⟩}
12         else if kind = U then
13           isQuiescent ⟵ false
14           outputs ⟵ outputs ∪ {⟨l, ⊥⟩}
15           destmap[g][⟨x̄, φ⟩] ⟵ destmap[g][⟨x̄, φ⟩] ∪ {⟨ρ, p'⟩}
16         else if kind = τ then
17           ⋮
18       if isQuiescent then
19         outputs ⟵ outputs ∪ {⟨⟨δ, (), true⟩, ⊥⟩}
20         destmap[δ][⟨(), true⟩] ⟵ destmap[δ][⟨(), true⟩] ∪ {⟨id, p⟩}
21     ⋮
22   ⋮
```

- in lines 11, 15, and 20 assignments to *destmap* are changed to reflect the two-level mapping of field n_l, and the presence of the variable mapping— note that here notation like $destmap[g][\langle \overline{t}, \varphi \rangle]$ does not refer to application of substitution, but to array (map) indexing;

This concludes the changes to method unfold.

Ad 5: method n In the non-symbolic setting of Chapter 4, Table 4.4 showed that method $n(l)$ just looks up the given label l in map $P.n_l$, and returns the associated set of LTS states. This suffices in the non-symbolic setting, because l matches *at most one* label in the map.

In the symbolic case the situation is different, as shown in Algorithm 6.2— note that also here square brackets denote map indexing, not substitution ap-

Algorithm 6.2: $P.n(l)$ for PTS Primer

```
   input  : P, a pseudo-state
            e, an Explorer that gives access to the Specification
            l = ⟨g, t̄⟩, an instantiated label
   output: S, the set of states, reachable from P via a transition with label l
 1 begin
 2     S ⟵ ∅
 3     m ⟵ P.n_l[g]  // P.n_l holds destmap as computed in Algo. 6.1
 4     foreach ⟨x̄, φ⟩ ∈ keys(m) do
 5         ρ, p' ⟵ m[⟨x̄, φ⟩]
 6         if |x̄| = |t̄| and |x̄| = 0 then
 7             S ⟵ S ∪ {p'}
 8         else if |x̄| = |t̄| and |x̄| > 0 then
 9             p'' ⟵ e.inst(p', ρ(x̄) ← t̄)
10             if p'' ≠ ⊥ then
11                 S ⟵ S ∪ {p''}
12     return S
```

plication. When the potential behaviour is represented using parameterized labels, a single instantiated label l may match *multiple* parameterized labels. Thus, without the two-level mapping, we should try to match l with all labels that represent the potential behaviour, i.e. all labels in the map. With the two-level mapping, we still have to match $l = \langle g, \bar{t} \rangle$ with all of those labels in the map that have the same gate g—hence the loop (at lines 4–11) over all tuples $\langle \overline{x}, \varphi \rangle$ that are key in the second level of the map. (Function *keys* returns the keys of second-level map m).

DerivationEngine interface implementation

No changes are necessary to the DerivationEngine interface implementation: all changes necessary have been taken into account, in the changes that we made to the pseudo-state type. Thus, also for the symbolic case, Table 4.5 shows how the Primer implements the DerivationEngine interface functions for the specification-only DerivationEngine (as long as the symbolic pseudo-state type is used).

6.6.4 Extension of Manager Algorithm

We have to extend Manager Algorithm 3.1 in three places, as shown in Algorithm 6.3:

1. the check whether the previous test step was an observation of quiescence at line 5 now uses the act function;
2. the application of a stimulus, at lines 10–17 where we use the Instantiator to obtain an instantiation of a parameterized label—when the instantiation fails, we set the verdict to "error";

Algorithm 6.3: Random On-Line Testing with PTS — changes w.r.t. Algo. 3.2

```
   input : A DerivationEngine d that gives access to the Specification, an
           Instantiator i, and an Adapter a that gives access to the SUT
 1 begin
 2     ⋮
 3     while v = ⊥ and the user does not stop the testing do
 4         ⋮
 5         else if act(prevLabel) = δ or ¬d.hasOutputs(P) then
 6             ⋮
 7         ⋮
 8         if action = stimulate then
 9             pick stimulus ⟨l′, V⟩ ∈ stimuli
10             if var(l′) ≠ ∅ then
11                 l″ ⟵ i.getInstance(l′, V)
12                 if l″ = ⊥ then
13                     v ⟵ error(unable to instantiate)
14             else
15                 l″ ⟵ l′
16             if v = ⊥ then
17                 t, l ⟵ a.tryStim(l″)
18         else obtain and check observation
19             ⋮
20         if v = ⊥ then
21             ⋮
22     ⋮
```

3. the invocation of next, and the evaluation of its result, is now guarded by a test that the verdict is still unset, to avoid invoking next when no interaction with the SUT took place because the instantiation failed (lines 20–21).

These are the only changes necessary.

6.6.5 Example

In Figure 6.9 we illustrate the interaction between Manager, Primer, Explorer and Adapter for the music player. In particular, we show how the invocation of next function of the Primer, with an instantiated label that matches a parameterized label, triggers the invocation of the inst function of the Explorer with the binding that results from the match.

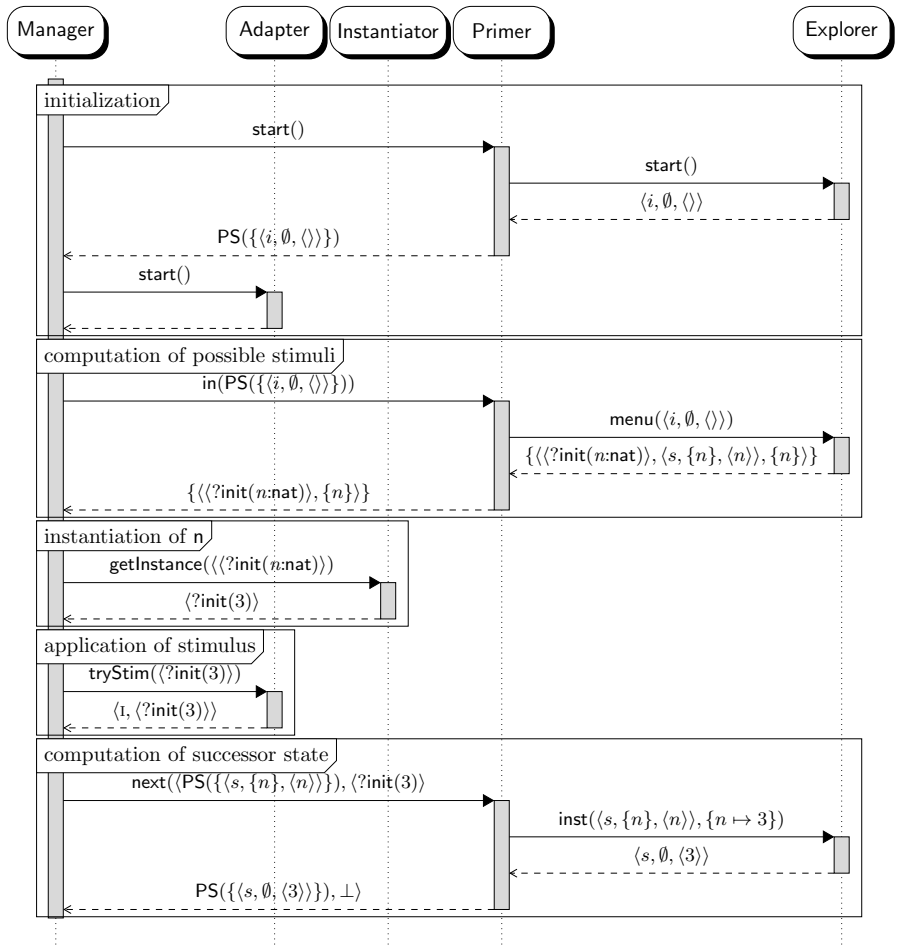


Figure 6.9: Sequence diagrams illustrating interaction between Manager, Primer, Explorer and Adapter for the music player. We give a simplified view on the manipulation of the pseudo-state: we do not show that it is actually *unfold* that invokes *menu* to initialize the pseudo-state fields i, o, and n_l, and thus pre-compute the results for in, out etc. Of the pseudo-state we only show the set of PTS states that it contains, and of those states we only show the fields that are relevant here, i.e. location, instantiation obligation, and the value of *nsong* (the value of *cur* is not relevant in this sequence).

6.6.6 Implementation Notes

In this chapter we have chosen to use the STS formalism as vehicle to explain our symbolic extensions. We represented the PTS formalism as an extended STS to avoid confusing the reading with additional notation and formalisms.

Our implementation of the symbolic extensions extends beyond what we described, to allow for "richer" modelling languages; it is inspired by the ETS

Table 6.4: Overview of symbolic Explorer implementation instances.

nr	name	main author	publications	case studies
1	SmileExp	Axel Belinfante	-	-
2	spex	Jeroen van Yperen	[vY07]	-
3	ta2torx	Henrik Bohnenkamp	[BB07, BB05]	Sec. 8.1.5
4	STSimulator	Lars Frantzen	-	[vSY13]

formalism [Eer94]. In particular, the parameterized labels in our implementation consist of tuples $\langle g, \bar{t}, \varphi \rangle$, where each t_i in $\bar{t}$ is a term in normal form. Our Explorer implementations use a naming scheme for the variables in the parameterized labels, to facilitate the pre-determinization, by avoiding that we have parameterized labels that only differ in variable names. In the Primer, we check whether an instantiated label is a candidate instance of a parameterized one, by matching the terms of instantiated labels with the terms of parameterized ones. When such matching succeeds, the result is a valuation for the variables in the parameterized label; such valuation is then used with the Explorer next function.

Implementations In Table 6.4 we give an overview of symbolic Explorer implementation instances.

SmileExp uses Smile [Eer94] to explore the transition system of a LOTOS model symbolically. It was developed to study the feasibility of testing with an symbolic Explorer; it was not used in practice.

spex uses Spin [Hol03] to explore the transition system of a Promela model symbolically. It was developed as a master thesis; it has been used at model-based testing company Axini [axi].

ta2torx gives access to the (timed) transition system of a network of timed automata. It has been used in a case study to test a Myrianed protocol entity, see Section 8.1.5.

STSimulator gives access to the transition system of an STS. It has been used in case studies, see [vSY13].

6.7 Timed Testing with a PTS

In [BB05, BB07] we discuss the extension of our tool to timed testing. Here we give an overview of our approach. With the time-aware ta2torx Explorer (see below) we test with $\mathbf{tioco}_M$.

Also for timed testing, we use a PTS as underlying formalism. In this PTS, each transition label has a parameter that represents time. A constraint on this time parameter gives the time-interval in which the interaction, represented by the label, should take place. The time-interval just consists of an lowerbound and an upperbound. The upperbound can be infinity.

For each interaction with the SUT, the Adapter creates a label representation—either as acknowledgement of the stimulus that was applied, or as representation

of an observation. In these instantiated labels, the time parameter has been instantiated such that it is a timestamp for the moment at which the interaction took place.

After each interaction, as part of the work done for the next function, the timestamp is fed back to the Explorer, such that observations can be checked, and, such that it can be checked whether a stimulus was applied in time: it may be the case that a stimulus is applied too late, in which case there may not be successor behaviour, and the test run stops with a verdict that indicates the reason for the premature end of the test run.

Timed-testing with our architecture To adapt our architecture for timed testing, we need

1. a time-aware Explorer,
2. a time-aware Instantiator, and
3. a time-aware Adapter.

Ad 1: time-aware Explorer The time-aware Explorer provides access to a PTS that contains time parameters in the labels. The ta2torx Explorer, mentioned in Section 6.6.6, and discussed in [BB05, BB07], uses a network of timed automata as model.

Ad 2: time-aware Instantiator The time-aware Instantiator just picks a random moment in the time-interval. To avoid that it chooses a moment in time that has already passed, it consults the system clock whenever it has to instantiate. To be able to deal with an upperbound of infinity, it is configured with a maximum delay.

Ad 3: time-aware Adapter For stimuli, the time-aware Adapter is given an instantiated label. It consults the system clock, to decide how long it must wait before trying to apply the stimuli. When the stimulus has been applied, the Adapter returns a label to acknowledge the application of the stimulus, like in the un-timed case. However, the Adapter does not return the label exactly as it was given: in the label that it returns, the time parameter has been adjusted to represent the moment at which the stimulus was applied. In this way, (via the Primer) the Explorer is informed when the stimulus was applied.

Also for observations, the Adapter returns a label, in which the time-parameter represents the moment at which the observation was received.

The JTorX builtin Adapter instances for toy implementations have a time-aware mode, which can be enabled in the JTorX GUI.

Application We used JTorX with the ta2torx Explorer in a case study to test a Myrianed protocol entity, see Section 8.1.5.

This concludes our description of timed testing.

6.8 Summary

In this chapter we introduced the APTS formalism, and showed how we can use it in our architecture to support symbolic models.

Chapter 7

Model-based specification, implementation and testing of a software bus

In this chapter we describe a medium-sized project where JTorX has been used effectively, namely the development of a software bus, called the XBus. The XBus was developed using formal engineering, i.e. by using formal methods during the design, implementation and testing phases of the development; model-based testing with JTorX was one of multiple techniques used.

The experiences that we report on were obtained during two phases: a first phase—an internship carried out at Neopost Inc. in the summer of 2009, and a second, post-internship analysis phase that took place at the university. In the first phase the XBus was developed, an mCRL2 model was created and simulated (Reqs 18, 19, 20), and used for on-line model-based testing of the XBus implementation (Reqs 4, 12, 17, 23, 24). In the second phase we performed model checking of the XBus protocol (Req 20), and measured the quality and performance of the model-based testing process (Reqs 23, 24). In both phases, the models used for model-based testing had an unbounded state space (Req 6); for model-checking, finite versions of these models were used.

Bibliographical note This chapter is derived from [SBSM14]. In the description of the work done during the first phase, i.e. during the internship at Neopost, we typically write "we", even when that work was carried out by the person doing the internship: Marten Sijtema, at that time a Computer Science MSc. student, and the first author of the related papers [SSBM11, SBSM14].

7.1 Introduction

Formal methods refer to a rich palette of mathematically rigorous modelling, analysis and testing techniques, including formal specification, model checking,

theorem proving, extended static checking, run-time verification, and model-based testing. The central claim made by the field of formal methods is that, while it requires an initial investment to develop rigorous models and perform rigorous analysis methods, these pay off in the long run in terms of better, more maintainable code. While formal engineering has been a success in large and safety-critical projects [HKL+10, FGM+10, KR10, Lev00, LK00], here we investigate this claim for this more modest and non-safety-critical project: model-based specification, implementation and testing of the XBus.

7.1.1 First phase: Developing the XBus

The XBus Neopost Inc. is one of the largest companies in the world producing supplies and services for the mailing and shipping industry, like franking and mail inserting machines, and the XBus is a software bus that supports communication between mailing devices and software clients. The XBus allows clients to send XML-formatted messages to each other over TCP (the X in XBus stands for XML), and also implements a service-discovery mechanism. That is, clients can advertise their provided services and query and subscribe to services provided by others.

We have developed the XBus using the classical V-model [Roo86] (see Fig. 7.1 on page 221), and used formal methods during the design and testing phase. The total running time of this project—i.e. of the internship—was 14 weeks. An important step in the design phase was the creation of a behavioural model $\mathsf{m}_{\mathsf{dev}}$ of the XBus, written in the process algebra mCRL2 [GKM+08, mCR]. We chose mCRL2 because of its powerful data types and function declarations, which turned out to be very helpful for our purpose. Model $\mathsf{m}_{\mathsf{dev}}$ pins down the interaction between the XBus and its environment in a mathematically precise way. We simulated the model to check its validity, which greatly increased our understanding of the XBus protocol and made the implementation phase a lot easier. Due to time-constraints we did not use model-checking during XBus development in this phase (model-checking was used in the second phase, as we discuss in Section 7.1.2).

Testing the XBus After implementing the protocol, we tested the implementation, i_1, distinguishing between data and protocol behaviour. *Data behaviour* concerns the input/output behaviour of a function and is static, i.e., independent of the order of methods calls. *Protocol behaviour* relates to the business logic of the system, i.e. the interaction between the XBus and its clients. Here, the order in which protocol messages occur crucially determines the correctness of the protocol. Therefore, we used unit testing to test the data behaviour and model-based testing for the protocol behaviour.

Model-based testing with JTorX We used JTorX to test the implementation against mCRL2 model $\mathsf{m}_{\mathsf{dev}}$. During the design phase, we already catered for model-based testing: we designed for testability by taking care that at the model boundaries, we could observe meaningful messages. Moreover, we made sure that the boundaries in the mCRL2 model matched the boundaries in the

architecture. To be able to connect JTorX to the implementation, we had to write an adapter (see Chapter 5). The adapter translates between the protocol messages from the mCRL2 model and physical messages in the implementation. Keeping the adapter simple was an important design decision for us. We achieved this by keeping a close correspondence between model $\mathsf{m}_{\mathsf{dev}}$ and the system architecture. Again, our design for testability greatly facilitated the development of the adapter.

After unit testing and repairing the issues uncovered by it, we ran JTorX (in random on-line testing mode) against implementation i_1 and mCRL2 model $\mathsf{m}_{\mathsf{dev}}$ (once configured, JTorX runs completely automatically) and found five subtle bugs. We believe that it is much harder to discover these bugs with unit testing, because they involve the order in which protocol messages should occur. After repairing them, we ran JTorX several times for more than 24 hours, without finding any more errors in i_1 (in the second phase, we found errors in i_2, the implementation that was developed and tested in that phase, see Section 7.1.2 Ad 3). After an acceptance test, the XBus was released for use in Neopost. Unfortunately, we do not have information about Neopost' experience with the XBus.

7.1.2 Second phase: Analysis

The development of the XBus, carried out in the first phase, supported the central claim made by Formal Methods—the use of rigorous methods during the development is cost effective. Nevertheless, it left room for questions: how thorough was the process carried out in 14 weeks? How good was the model—this is important, since the model-based test process is as good as the model. Would model checking have helped to produce better code? Was the testing thorough enough; what can we say about coverage? In the second phase, we investigated these questions. In particular, we focused on (1) the added benefits of model checking, (2) the quality of the models, and (3) test coverage.

Ad 1: The added benefits of model checking We started out by model checking the model $\mathsf{m}_{\mathsf{dev}}$ that was created during the development phase. We created a series of new models with different features. Firstly, we needed to change $\mathsf{m}_{\mathsf{dev}}$ to make it finite, yielding the model $\mathsf{m}_{\mathsf{dev,fin}}$: $\mathsf{m}_{\mathsf{dev}}$ allows an unbounded number of client connections, and uses arbitrary integers as connection identifiers—this is not a problem for (on-line) testing, but for model checking it is, as it leads to an infinite state space.

We used evaluator4 [CAD12] from the CADP toolset to model check the XBus requirements obtained during the first phase; these requirements were formalised in the logic MCL [MT08], which is an extension of the μ-calculus with data. The main reason that we chose evaluator4 is that it allows reasoning over the individual parameters of the messages (labels). We used the mCRL2 and LTSmin toolsets to obtain, from the mCRL2 model, the binary coded graph (`.bcg`) file that evaluator4 needs. Model checking did not uncover errors in model $\mathsf{m}_{\mathsf{dev,fin}}$. With hindsight, this is not so surprising, because model $\mathsf{m}_{\mathsf{dev}}$ had been used extensively already in simulation and model-based testing, and because

of the simple structure of the model, which we describe in Section 7.4.1 on page 228. We did find $\mathsf{m}_{\mathsf{dev,fin}}$ (and hence $\mathsf{m}_{\mathsf{dev}}$) to be incomplete. In particular, those requirements related to so-called bad weather behaviour are not present: invalid or unexpected messages were not modelled, and neither were empty lists of services.

Compared to testing, the model checking process was very labor intensive: i.e. reworking the models, formalising requirements, playing round with tricks to reduce the state space and making sure that the time needed for model checking was manageable. This took us 4 person weeks. The entire model-based testing approach, i.e. writing the model $\mathsf{m}_{\mathsf{dev}}$, creating the adapter, executing and analysing the tests, took 3 person weeks.

Ad 2: Model quality We included the additional requirements that we uncovered during our model checking activities and created a family of models, all in mCRL2, as shown in Fig. 7.5 on page 231. For each model, we constructed an infinite variant (for testing) and a finite one (for model-checking). Also, we investigated the use of queues: model $\mathsf{m}_{\mathsf{dev}}$ is simple in the sense that it neither models the incoming message queue, nor the fifo-queue-like behaviour of the TCP connections between the XBus server and its clients. We show that including these queues in the model does not affect test coverage, but does significantly increase testing time. Finally, we constructed several model variants that were more liberal wrt the accepted inputs. An overview of this family of models is given in Section 7.4.2.

Ad 3: Test coverage Code coverage metrics [Mye79] are standard measures to evaluate the quality of a test suite: the higher the coverage, the more faults a test suite can potentially find. We extensively evaluated the thoroughness of our testing, by measuring code coverage and model coverage. Since the original XBus implementation (i_1) is proprietary software of Neopost, we had no longer access to it after the internship. Therefore, we used i_2, a carefully reconstructed implementation. We used branch coverage as our code coverage metric, i.e. the percentage of all branches in the control flow graph that were executed during testing. To do so, we instrumented the code of the (reconstructed) implementation by hand. For model coverage, we used the percentage of *linear process specification* (LPS) summands executed. LPSs are a uniformized representation of mCRL2 models. Complete LPS coverage basically means that each nondeterministic alternative is executed at least once.

We have extensively analysed model and code coverage from short test runs (10,000 test steps, 5–30 minutes) and long ones (250,000 test steps, 2–40+ hours), with each of our (infinite) model versions. All the tests were derived fully automatically by doing a random walk over (the state space of) the model. We found that the maximal code coverage is typically already reached after 1000 test steps, i.e. after at most two minutes of testing.

We found that, the more complete a model was (wrt requirements or accepted inputs), the higher code coverage could be obtained.

7.1.3 Our findings

In the first phase, during the internship, writing the model, simulating it, and testing the implementation with JTorX only took 17% of the total development time. Therefore, we conclude that the formal engineering approach has been very successful: with limited overhead, we have created a reliable software bus with a maintainable architecture. Thus, as in [GVZ01], we clearly show that formal engineering is not only beneficial for large, complex and/or safety-critical systems, but also for more modest projects.

In the second phase, during the analysis, after the internship, we found that, within the limits of the model, the model-based testing that was done during the project was rather thorough. However, we also found that the model, used to derive these tests, was not fully complete, and more thorough analysis of the requirements, during the project would have been desirable. This could have been achieved with model checking, but at a high cost. We expect that more light-weight methods that trace the requirements in the model are more cost effective.

Finally, we experienced that model and code coverage metrics can provide valuable insight in the quality, effectiveness, and progress of the model-based testing process. Based on our experiences, we advocate that formal engineering pays off, and that investing in high-quality models is worth-while. Formal engineering pays off, in the sense that with limited overhead, a reliable software bus was created. Investing in high-quality models is worthwhile, because the quality of model-driven development lies within the quality of the model. To do so, we believe that models should be as complete as possible, i.e. accept all inputs and include all requirements. Extensive simulation and—though expensive—model-checking help. Also, we believe that measuring coverage is helpful: if less than 100% code coverage is achieved, then the model should be augmented.

Remainder of this chapter The remainder of this chapter is organised as follows. Section 7.2 provides the context of the XBus implementation project. Then, Section 7.3 describes the activities involved in each phase of the development of the XBus, including the activities done during the analysis after the project. Section 7.4 gives the details of modelling, creation of additional models, and model checking, and Section 7.5 gives the details of model-based testing, and of code coverage and model coverage analysis. Section 7.6 reflects on the lessons learn ed in this project. Finally, Section 7.7 presents conclusions.

7.2 Background

7.2.1 The XBus and its context

Neopost Neopost Incorporated [Neo09] is one of the world's main manufacturers of equipment and supplies for the mailing industry. Neopost produces both physical machines, like franking and mail inserting machines, as well as software to control these machines. Neopost is a multinational company headquartered in Paris (France) that has departments all over the world. Its

software division, called Neopost Software & Integrated Solutions (NSIS) is located in Austin, Texas, USA. This is where the XBus implementation project took place.

Shipping and franking mail Typically, the workflow of shipping and franking is as follows. To send a batch of mail, one first puts the mail into a folding machine, which folds all letters. Then an inserting machine inserts all letters into envelopes[1] and finally, the mail goes into a franking machine, which puts appropriate postage on the envelopes and keeps track of the expenses.

Thus, to ship a batch of mail, one has to set up this process, selecting which folding, inserting and franking machine to use and configure each of these machines, setting the mail's size, weight, priority, and the carrier to use. These configurations can be set manually, using the machine's built-in displays and buttons. More convenient, however, is to configure the mailing process via one of the desktop applications that Neopost provides.

The XBus To connect a desktop application to the various machines, a software bus, called the XBus, has been developed. The XBus communicates over TCP and allows clients to discover other clients, announce provided services, query for services provided by other clients and subscribe to services. Also, XBus clients can send self-defined messages across the bus.

When this project started, an older version of the XBus existed, called the XBus version 1.0. The goal of our project was to re-implement the XBus while maintaining backward compatibility, i.e. the XBus 2.0 must support XBus 1.0 clients. Key requirements for the new XBus were improved maintainability and testability.

7.2.2 The specification language mCRL2

The language mCRL2 [GKM+08, mCR] is a formal modeling language for describing concurrent systems, developed at the Eindhoven University of Technology. It is based on the process algebra ACP [BK85], and extends ACP with rich data types and higher-order functions. The mCRL2 toolset facilitates simulation, analysis and visualisation of behaviour; as we discussed in Section 4.2.4, it also contains the LPS2TORX tool (Explorer component) which enables model-based testing against mCRL2 models. Specifications in mCRL2 start with a definition of the required data types. Technically, the behaviour of the system is declared via process equations of the form $X(x_1 : D_1, x_2 : D_2, \ldots, x_n : D_n) = t$, where x_i is a variable of type D_i and t is a process term, see the example in Section 7.3.2. Process terms are built from (1) (potentially parameterised) actions; (2) operators: alternative composition, sum, sequential composition, conditional choice (if-then-else), parallel composition; and (3) encapsulation, renaming, and abstraction. Actions represent basic events (like sending a message or printing a file) which are used for synchronisation between parallel processes. Apart

[1] Alternatively, a combined folding/inserting machine can be used.

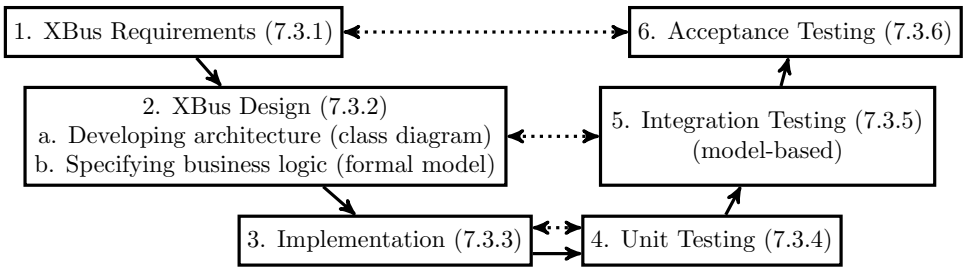


Figure 7.1: The V-model used for development of XBus; parenthesized numbers refer to sections.

from analysis within the tool set, mCRL2 interoperates with other tools: specifications in mCRL2 can be model checked via the CADP model checker by generating the state space in `.aut` or `.bcg` format, they can be proven correct using e.g. the theorem prover PVS, and they can be tested against with JTorX. For model checking, we used the evaluator4 tool from the CADP tool set. The tool evaluator4 is able to check whether a model-checking formula, given in its input language MCL, holds for (the state space generated from) an mCRL2 model m.

7.3 Development of the XBus and post case-study analysis

Below, we describe all the activities in the development (phase 1; during the internship) and analysis (phase 2; after the internship) of the XBus. We developed the XBus according to the classical V-model ([Roo86], see Fig. 7.1). For each step in the V-model we report the activities carried out in both phases—each section below corresponds to one step in the V-model, see Fig. 7.1 again.

During the development, the overall test strategy was to test data behaviour using unit testing, and to test protocol behaviour, i.e. the interaction between XBus and its clients, using model-based testing. We chose to use model-based testing for protocol behaviour, because here the dynamic behaviour, i.e., the order of protocol messages, crucially determines the correctness of the protocol. We only started with model-based testing of protocol behaviour after we had completed unit testing of data behaviour.

7.3.1 XBus requirements

First phase We obtained the functional and nonfunctional requirements by studying the documentation of the XBus version 1.0 (a four page English text document) and by interviewing the manager of the XBus development team.

The functional requirements express that the XBus is a centralised software application which can be regarded as a network router: clients can connect and disconnect at any point in time; connected clients can send XML-formatted

1.	XBus messages are formatted in XML, following the same Schema as the XBus 1.0.
2.	Clients connecting to XBus perform a handshake with the XBus server. The handshake consists of a $\mathsf{Conn}_{\mathsf{req}}$—$\mathsf{Conn}_{\mathsf{ack}}$—$\mathsf{Conn}_{\mathsf{auth}}$ sequence.
3.	Newly connected clients are assigned unique identifiers.
4.	Clients can subscribe to be notified when a client connects or disconnects.
5.	Clients can send messages to other clients with self-defined, custom, data. Such messages can have a self-defined, custom message type. In addition there are protocol messages for connecting, service subscription, service advertisement.
6.	Clients can subscribe to receive all messages sent by other clients that are of one or more given types (including self-defined messages), using the Sub message.
7.	Clients can announce services that they provide, using the $\mathsf{Serv}_{\mathsf{ann}}$ message.
8.	Clients can inquire about services, by specifying a list of service names in a $\mathsf{Serv}_{\mathsf{inq}}$ message. Service providers that provide a subset of the inquired services will respond to this client with the $\mathsf{Serv}_{\mathsf{rsp}}$ message.
9.	Clients can send *private* messages, which are only delivered to a specified destination.
10.	Clients can send *local* messages, which are delivered to the specified address, as well as to clients subscribed to the specified message type.

Table 7.1: Overview of XBus requirements obtained in the first phase.

11.	Invalid messages are discarded.
12.	Unexpected messages are discarded.
13.	$\mathsf{Serv}_{\mathsf{ann}}$, $\mathsf{Serv}_{\mathsf{inq}}$ and $\mathsf{Serv}_{\mathsf{rsp}}$ messages with an empty list of services are not broadcast to other clients.
14.	$\mathsf{Notif}_{\mathsf{local}}$ messages are not broadcast to source or destination of a $\mathsf{Local}_{\mathsf{req}}$ message.

Table 7.2: Overview of additional XBus requirements obtained in the second phase.

messages to each other. Moreover, clients can discover other clients, announce services, and query for services that are provided by other clients. Also, they can subscribe to services, and send self-defined messages to each other. Table 7.3 on the next page gives an overview of the XBus protocol messages. Table 7.1 summarises the functional requirements; important non-functional requirements are testability, maintainability and backwards compatibility with the XBus 1.0.

Second phase While formalising the requirements in Table 7.1 and model checking them on model $\mathsf{m}_{\mathsf{dev}}$, we realised that so-called bad weather behaviour was not present in $\mathsf{m}_{\mathsf{dev}}$ nor in the requirements. Therefore, we extended both the model, and the list of requirements. Table 7.2 shows the additional requirements, pinning down what to do with unexpected inputs.

7.3.2 XBus design

First phase In the first phase, the design step encompassed two activities: we created

Connection establishment and release		
$\mathsf{Conn}_{\mathsf{req}}$	*input*	(implicit) implied by a client establishing a TCP connection with XBus.
$\mathsf{Conn}_{\mathsf{ack}}$	*output*	sent from XBus to a client just after the client establishes a TCP connection with the XBus, as part of the handshake.
$\mathsf{Conn}_{\mathsf{auth}}$	*input*	sent from a client to the XBus to complete the handshake.
$\mathsf{Disc}_{\mathsf{req}}$	*input*	(implicit) implied by a client closing its TCP connection with XBus.
Service announcement and inquiry		
$\mathsf{Serv}_{\mathsf{ann}}$	*input*	sent (just after connecting) from a client c to XBus, which broadcasts it to all other connected clients, to announce the services provided by c.
$\mathsf{Serv}_{\mathsf{inq}}$	*input*	sent (just after connecting) from client to XBus, which broadcasts it to all other connected clients, to ask what services they provide.
$\mathsf{Serv}_{\mathsf{rsp}}$	*output*	sent from a client via XBus to another client, as response to $\mathsf{Serv}_{\mathsf{inq}}$, to tell the inquirer what services the responding client provides.
Event subscription and notification		
(Un)Sub	*input*	sent from a client to XBus, with as parameter a list of (custom) message types, to (un)subscribe receipt of all messages of the given types.
$\mathsf{Notif}_{\mathsf{conn}}$	*output*	sent from XBus to clients that subscribed to connect notifications.
$\mathsf{Notif}_{\mathsf{disc}}$	*output*	sent from XBus to clients that subscribed to disconnect notifications.
$\mathsf{Notif}_{\mathsf{local}}$	*output*	sent from XBus to clients that subscribed to non-private messages.
Messages to other clients		
$\mathsf{Local}_{\mathsf{req}}$	*input*	sent from client to XBus, to be delivered to indicated client (as $\mathsf{Local}_{\mathsf{ind}}$), and to other clients that have subscribed to the given message type (as $\mathsf{Notif}_{\mathsf{local}}$).
$\mathsf{Local}_{\mathsf{ind}}$	*output*	sent from XBus to clients, as consequence of a received $\mathsf{Local}_{\mathsf{req}}$.
$\mathsf{Priv}_{\mathsf{req}}$	*input*	sent from client to XBus, to be delivered to indicated client only.
$\mathsf{Priv}_{\mathsf{ind}}$	*output*	sent from XBus to clients, as consequence of a received $\mathsf{Priv}_{\mathsf{req}}$.

Table 7.3: Overview of XBus protocol messages.

(*A*) an architectural design, given by the UML class diagram in Fig. 7.2 on the following page, and

(*B*) an mCRL2 model, $\mathsf{m}_{\mathsf{dev}}$, describing the protocol behaviour.

We used the mCRL2 simulator to validate the design and model $\mathsf{m}_{\mathsf{dev}}$. As stated, time constraints prevented us to use model-checking in this phase.

The architectural design and mCRL2 model $\mathsf{m}_{\mathsf{dev}}$ were developed in parallel. Central in their design are the XBus messages: each message translates into a method in the class diagram and into an action in mCRL2 model $\mathsf{m}_{\mathsf{dev}}$. The UML diagram specifies which methods are provided, while the mCRL2 model $\mathsf{m}_{\mathsf{dev}}$ describes the order in which actions should occur, i.e. the order in which methods should be invoked. Thus, the architectural model in UML and the behavioural model in mCRL2 are tightly coupled and complementary.

Ad A: Architectural design The architecture of the XBus is given in Fig. 7.2 on the next page, and is based on a standard client-server architecture. Thus, the XBus has a client side, implemented by the XBusGenericClient, and a server side, implemented by the XBusManager. The latter handles incoming protocol messages and sends the required responses. Both the server and

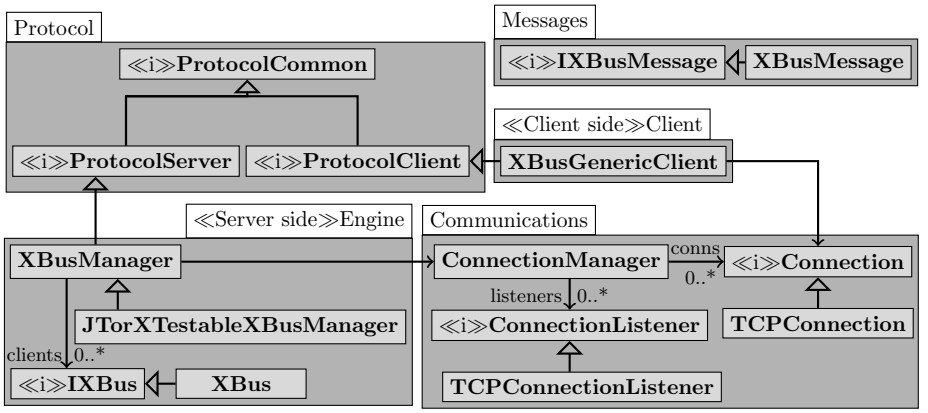


Figure 7.2: High level architecture of the XBus system. It contains a server side package, and a client side package. Furthermore, it has functionality for TCP connections and XBus messages. Both server and client implement the Protocol abstract class. All interfaces are indicated with ≪i≫.

the client use the Communications package, which implements communication over TCP. As illustrated in Fig. 7.3 on the facing page, the ConnectionManager class in the Communications package uses a queue data structure as a buffer for incoming messages. When a message is handed over from the Communications package to its user—in the server this user is the XBusManager—it is popped from the queue.

We catered for model-based testing already in the design: class XBusManager has a subclass JTorXTestableXBusManager. During testing, this subclass overrides the send message of class XBusManager, allowing JTorX to have more control over the state of the XBus server; see Section 7.3.5 for more details.

Ad B: The mCRL2 model We modelled the required XBus behaviour as an mCRL2 process. We profited from mCRL2's concise notation for enumerated types, records, and lists, and the ability to define functions.

A key decision in creating a model is what to model, and to determine the abstraction level and model boundaries. We chose to model the XBusManager, i.e. the handling of the messages that come into the server; this is the most critical part of the XBus functionality. Thus, the Communications package is not included in model $\mathsf{m}_{\mathsf{dev}}$, and neither are the internal components like the TCP-sockets, nor the queue that the Communications package uses as a buffer for incoming messages. Thus, for each message that arrives at the server, $\mathsf{m}_{\mathsf{dev}}$ models how to handle this message: it will either send a reply, relay, or broadcast the message. Then, $\mathsf{m}_{\mathsf{dev}}$ will update its internal state: in order to determine the correct response, the server keeps track of the client's state by keeping an internal list of client objects.

In Section 7.4.1 we discuss the model in more detail.

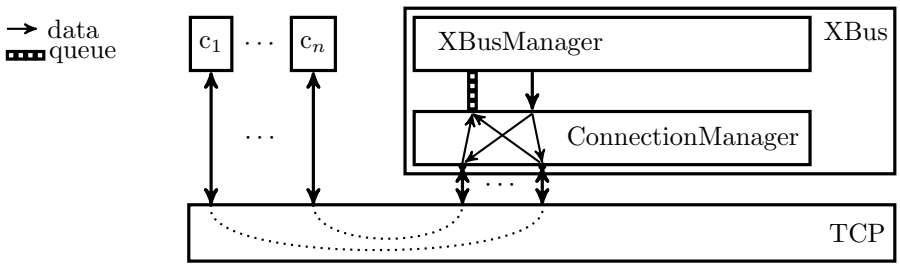


Figure 7.3: Communication between clients $c_1, \ldots, c_n$ and XBus. We show how the XBus is decomposed into XBusManager and ConnectionManager. Incoming messages are collected in a queue in the ConnectionManager—here drawn as the left connection between XBusManager and ConnectionManager. The XBusManager processes these messages one by one; it sends responses using methods offered by the Communications package—this is represented by the arrow from XBusManager to ConnectionManager.

Second phase In the second phase, we evaluated the quality of model $\mathsf{m}_{\mathsf{dev}}$, where we looked at both completeness and correctness—using model checking—of the model. When we found that model $\mathsf{m}_{\mathsf{dev}}$ was incomplete—i.e., not all requirements are represented in it—we created additional models, and used model-checking to check their correctness. We elaborate on these activities, and on the additional models, in Section 7.4.2.

7.3.3 Implementation

First phase In the first phase, we created implementation i_1, at Neopost, for use by Neopost. Implementation i_1 was only created once we had sufficient confidence in the quality of the design—to a large extent due to modelling and simulation. As mentioned in Section 7.1.1, we kept a close correspondence between model $\mathsf{m}_{\mathsf{dev}}$ and the system architecture. The programming language used was C#—use of .NET is Neopost company policy. Together with XBus server i_1, also an XBus client library was implemented, to ease construction of XBus clients. As we will see in Section 7.3.5, this client library was also used during model-based testing of XBus server i_1.

Second phase Since implementation i_1 is proprietary software of Neopost, it was not available during the second phase. Therefore, we carefully created a second implementation, i_2, to allow analysis of the thoroughness of model-based testing. Implementation i_2 was written in the programming language Go [The12]. Implementation i_2 has the same functionality as i_1, except that i_2 uses labels from the model, rather than XML formatted messages.

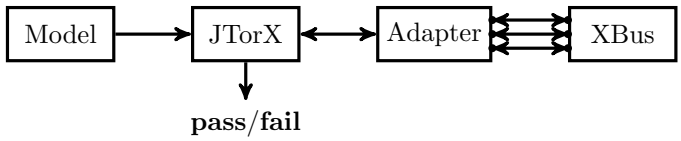


Figure 7.4: Testing XBus with JTorX playing the role of 3 clients.

7.3.4 Unit testing

First phase In the first phase, implementation i_1 was tested using unit tests as described below. Because the overall test strategy was to test data behaviour using unit testing, and to test protocol behaviour using model-based testing, the classes in the Communications and Messages packages were tested using unit testing; details are out of scope for this thesis. For the Communications package, unit tests were written to test the ability to start a TCP listener and to connect to a TCP listener, to test the administration of connections, and to test transfer of data. For the Messages package, unit tests were written to test construction, parsing and validation of messages. The latter was tested using both correct and incorrect messages.

Each error that was found during unit testing was immediately repaired.

Second phase In the second phase, no unit tests were run—implementation i_2 was only tested using model-based testing.

7.3.5 Model-based integration testing

For both implementations, we used model-based testing for the business logic, i.e. to test the interaction between XBus and its clients. In the first phase, for implementation i_1, we did this after we had completed unit testing of data behaviour.

General test set up To test whether the XBus implementations interact correctly with their environment, we first have to decide on a test set up. In both phases, we used the same test set up with three XBus clients, see Fig. 7.4 (although, as we show, the chosen test architecture differed). Three XBus clients is the smallest number that allows testing interesting scenarios that involve multiple clients. Thus, JTorX plays the role of three XBus clients, which are able to perform all protocol actions described in Section 7.3.1.

Typically, such scenarios require one client to trigger the activity—for example by connecting or disconnecting, or by sending a $\mathsf{Serv}_{\mathsf{inq}}$ message. A second client is necessary to cooperate in the activity, i.e. to witness or to realise the effect—by receiving a message and, possibly, responding to it. The third client can be used either to show the effect on a client that does not cooperate in the activity, or to show that the XBus is correct when multiple clients do cooperate in the activity. Typically, a single test run contains (many) instances of either of these roles for the third client.

It is our experience that model-based testing can easily generate long test runs, in which, at least for models that are as small as the one in this project, each possible scenario that can take place, does take place, multiple times. We come back to this in the discussion of (code- and model) coverage, in Section 7.5.

Dealing with potential message reordering As mentioned above, what we modelled is the XBusManager. However, the XBusManager implementation that we want to test is just one component of the XBus server. So, as we have seen more often when applying model-based testing (see e.g. Section 4 of [BFdV+99]), we could not connect the test tool directly to the implementation that we wanted to test (the XBusManager), at interfaces that coincide with the model boundaries. An obvious way to test the XBusManager, is via the XBus server in which it is contained, and interact with the XBus server via TCP connections—one for each XBus client impersonated by JTorX that has a connection to the XBus. However, messages that are sent at approximately the same moment, in the same direction, over different TCP connections between the XBus and its clients (whether impersonated or not), may overtake each other.

For *stimuli*, JTorX is in control: it can, if necessary, reduce the rate at which stimuli are sent to the point that, when the next stimulus is sent, the previous one will already have been received by the XBus. In both phases we just assumed that, compared to the network, JTorX is slow, such that the pace at which JTorX sends stimuli is slow enough to avoid one stimulus overtaking another one.

We used different solutions to deal with the possibility that *responses* would overtake each other—if we would not have dealt with this possibility, the tester might have emitted a **fail** verdict to a sequence of responses whose order was scrambled by the TCP channel. In the first phase, we extended the XBus implementation with an additional interface that provided JTorX access to the responses in the order in which the XBusManager produced them. In the second phase, instead, we relaxed the model, to not only accept the responses to a single stimulus in the single order in which they were produced by the XBusManager, but also accept any possible reordering. We discuss details in Section 7.5.

First phase After unit testing had been completed, and all errors that were found had been repaired, we tested implementation i_1 against model $\mathsf{m}_{\mathsf{dev}}$ using JTorX, to find errors. We found 5 bugs. Typically, a bug was found within 5 minutes after the start of a test. All these bugs concern the order in which protocol messages must occur. Therefore, it is our firm belief that such bugs are much harder to discover with unit testing. After these bugs had been repaired, we ran JTorX several times for more than 24 hours, without finding any more errors in i_1 (later, we found a few more bugs when we tested i_2—recall that i_2 is a separate implementation). In Section 7.5.1 we discuss the details of the test architecture that we used, and of the bugs that we found.

Second phase In the second phase, we tested implementation i_2 against all (infinite) models that we created in that phase, not in order to find errors,

but to investigate the thoroughness of the testing process, by looking at code coverage and model coverage. We did not test i_2 against model $\mathsf{m}_{\mathsf{dev}}$, because the different solution (for responses that might overtake each other) led to a different test architecture than in the first phase, one that was not consistent with $\mathsf{m}_{\mathsf{dev}}$. Among the models that we did test i_2 with, though, is model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$: this model was derived from $\mathsf{m}_{\mathsf{dev}}$, by extending it to cater for the slightly different test architecture. We ran tests of 10,000 steps, and of 250,000 steps, fully automatically.

Regarding code coverage, we found that with all models the maximal coverage was already reached in the test runs of 10,000 steps. With model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ we obtained 79% code coverage. This is no surprise: we know that $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ does not contain all possible messages, and thus certain stimuli can not be generated from it. With the most complete model, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, we obtained 100% code coverage. The coverage obtained with the other models was between these two numbers. Regarding model coverage, we saw that with each model we reached the maximal coverage possible, in the runs of 250,000 steps. However, we needed many more test steps to reach maximal model coverage, than to reach maximal code coverage.

In Section 7.5.2 we discuss the details of the test architecture that we used, and the test that we ran; in Sections 7.5.3–7.5.5 we discuss in more detail the coverage that we obtained; and in Section 7.5.6 we discuss the test execution time.

7.3.6 Acceptance testing

First phase Acceptance testing was done in the usual way: we organised a session with the manager of Neopost's ISS group, and showed how the XBus 2.0 implementation worked. In particular, we demonstrated that it implements the features required in Section 7.3.1.

Second phase In the second phase no acceptance testing was performed.

7.4 Modelling & Model Checking of the XBus

This section zooms in on the modelling and model checking activities described in Section 7.3.2.

7.4.1 The model $\mathsf{m}_{\mathsf{dev}}$

As mentioned, the mCRL2 model $\mathsf{m}_{\mathsf{dev}}$ describes the desired functioning of the XBusManager package, which is responsible for the handling of XBus messages and therefore the most central part of the XBus. Internally $\mathsf{m}_{\mathsf{dev}}$ keeps track of the state of all connected clients. Based on this state $\mathsf{m}_{\mathsf{dev}}$ decides, when a message arrives, how to handle it: send a reply or broadcast, relay it, or simply ignore it. After handling the message, $\mathsf{m}_{\mathsf{dev}}$ updates its internal state.

```
1 proc listening(c:Clients) =
2    (sum j:Int.(j >= 0 && j < numClients(c) &&
3                getClientStatus(j, c) == DISCONNECTED )
4       -> (ConnectRequest.ConnectAcknowledge.
5           listening(changeClientStatus(j, c, AWAIT_AUTH)))
6       <> delta
7    ) + ...
```

Listing 1: Definition of XBus handling of $\mathsf{Conn}_{\mathsf{req}}$ message in mCRL2.

Data Model $\mathsf{m}_{\mathsf{dev}}$ stores its internal data in a single data object: a list of clients, modelled as a list of data structures. For each client, the following information is kept.

- an integer that represents the identity of the client;
- the connection status of the client, being either: disconnected, awaiting-Authentication, or connected;
- the subscriptions of the client, which is a list of message types.
- the services that the client provides, which is a list of integers.

Behaviour Model $\mathsf{m}_{\mathsf{dev}}$ consists of a single process that operates in the following loop: (1) accept a message, (2) send zero or more responses, (3) update the internal state, i.e., the client list. After these steps, $\mathsf{m}_{\mathsf{dev}}$ is ready to process the next message. For example, when $\mathsf{m}_{\mathsf{dev}}$ receives a $\mathsf{Conn}_{\mathsf{req}}$, it replies with a $\mathsf{Conn}_{\mathsf{ack}}$, and adds the new client to the client list.

Listing 1 shows a (slightly simplified) part of $\mathsf{m}_{\mathsf{dev}}$. The process is named listening, and has as single parameter the list of clients c. The listing shows that from each client j that currently is in disconnected state (line 3), the server is willing to accept a $\mathsf{Conn}_{\mathsf{req}}$ message, after which it will send out a $\mathsf{Conn}_{\mathsf{ack}}$ message (line 4). Then it will update the status of the j^{th} client in the list and continue processing via a recursive call (line 5).

Model size The entire model consists of 6 pages (180 lines, 12kB) of mCRL2, including comments (without comments and blank lines: 142 lines, 9kB). Approximately half of it concerns the specification of data types and functions over them; the other half is the behavioural specification.

Model validation During the construction of the model, we intensively used the simulator from the mCRL2 toolkit. We incrementally simulated smaller and larger models, using both manual and random simulation. This was done for two reasons. First, to get a better understanding of the working of the whole system, and to validate the design already before the implementation activity was started. This was particularly useful to improve our understanding of the XBus protocol, of which only a (non-formal) English text description was available, which contained several ambiguities. Second, to validate the model,

to be sure that it faithfully represents the design, such that when we use JTorX to test our implementation against the model, all tests that JTorX derives from the model will yield the correct verdict.

7.4.2 Model checking & model transformation

Model completeness During the analysis, we carefully studied the requirements, and tried to formalise and model check them on model $\mathsf{m}_{\mathsf{dev}}$. We found that model $\mathsf{m}_{\mathsf{dev}}$ is incomplete, in the sense that not all requirements are represented in it. Model $\mathsf{m}_{\mathsf{dev}}$ does not contain self-defined messages, (i.e. no private or local messages) and thus requirements 5 and 6 can only be checked partially, and requirements 9 and 10 can not be checked. Also, $\mathsf{m}_{\mathsf{dev}}$ does not model lists of services, but only uses a singleton list with exactly one service, and thus requirements 7 and 8 can only be checked partially. Finally, $\mathsf{m}_{\mathsf{dev}}$ does not consider the formatting of the messages, and thus requirement 1 can not be represented in it.

During this analysis we also found that the list of requirements was incomplete: it only dealt with good-weather behaviour; bad-weather behaviour was left unspecified. Thus, requirements 11–14 were added during this analysis.

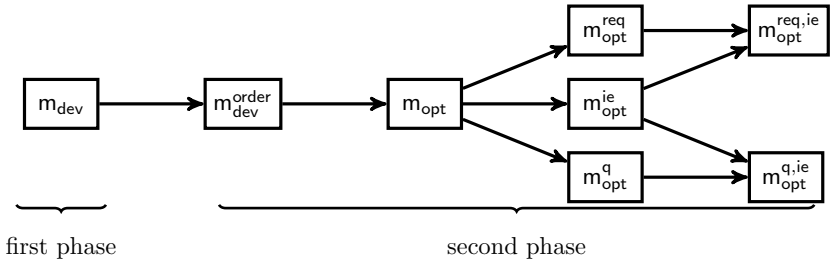
Family of models In order to incorporate the missing behaviour, we created a family of mCRL2 models, see Figure 7.5, that incorporate all requirements except for requirement 1: we do still not consider the XML formatting of the messages. We added the following two sets of features: (1) self-defined, private and local messages, and non-empty lists of services (instead of a singleton list with one element), (2) empty lists of services, invalid messages, and other bad weather behaviour. We did this in several steps, to control the amount of change introduced by each step, and to be able to observe (and show) the impact of the change on state space size (see Table 7.4 on page 232) and testing speed and coverage (see Sections 7.5.3–7.5.6).

We started with $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, which is the same as $\mathsf{m}_{\mathsf{dev}}$ except that it caters for the test architecture from the second phase. We optimised this model to $\mathsf{m}_{\mathsf{opt}}$, in which each state has a unique representation. From $\mathsf{m}_{\mathsf{opt}}$, we investigated three different variants: $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$ extends the requirements with the first set of features mentioned above; $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$ is an (semi) input-enabled variant of $\mathsf{m}_{\mathsf{opt}}$, i.e. when it is ready to accept input, it accepts any input; finally, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ is obtained from $\mathsf{m}_{\mathsf{opt}}$ by adding a message queue. We combined $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$ into $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$. Similarly, we combined $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ into $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$. As explained below, each model comes in two variants: an infinite one for testing, and a finite one—indicated via a subscript fin—for model checking.

Finite state space variants The original model $\mathsf{m}_{\mathsf{dev}}$ was infinite: it could accept an unbounded number of clients, where it used an unbounded integer as connection identifier in the protocol messages. For on-the-fly testing this is not a problem, because we only generate the portion of the state space that the system is currently in. For model checking, however, we need the complete state

$\mathsf{m}_{\mathsf{dev}}$	the original model, created in the first phase, during development of the XBus. This model alternates between accepting an input and producing the corresponding outputs, and it is not input-enabled: for example, after a client has sent a Sub message for a certain event e, a subsequent Sub message for e is only accepted after an Unsub message for e.
$\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$	obtained from $\mathsf{m}_{\mathsf{dev}}$, with one very small change, to accommodate the slightly different test architecture that we used in the second phase, as discussed in Section 7.3.5: it allows all possible interleavings of the outputs produced for a single input.
$\mathsf{m}_{\mathsf{opt}}$	an optimized version of $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, in which each state has a unique representation (finite state space variant of $\mathsf{m}_{\mathsf{opt}}$ is branching bisimilar to finite state space variant of $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$).
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$	obtained from $\mathsf{m}_{\mathsf{opt}}$, (semi) input-enabled: when the system accepts input, all (known) inputs are allowed.
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$	obtained from $\mathsf{m}_{\mathsf{opt}}$, by extending it such that all requirements (except requirement 1) are represented (but without input enabling, and: no invalid messages, and no messages with an empty list).
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$	derived from $\mathsf{m}_{\mathsf{opt}}$, by extending it such that all requirements (except requirement 1) are represented, and making it (semi) input-enabled.
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$	obtained from $\mathsf{m}_{\mathsf{opt}}$, by adding a queue context (but without input enabling, and without extending it to represent additional requirements).
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$	obtained from $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$, by making it (semi) input-enabled.

(a)



(b)

Figure 7.5: Overview of models discussed in this chapter (a) and relation between those models (b).

model	*#states*	*#transitions*	*#labels*
$\mathsf{m}_{\mathsf{dev,fin}}$	4,198,090	21,476,661	71
reduced (strong bisimulation)	686,151	1,236,486	71
reduced (branching)	362,958	867,666	71
$\mathsf{m}^{\mathsf{order}}_{\mathsf{dev,fin}}$	8,129,310	30,217,652	71
reduced (strong bisimulation)	198,095	425,112	71
reduced (branching)	83,414	194,271	71
$\mathsf{m}_{\mathsf{opt,fin}}$	133,857	1,019,196	71
reduced (strong/branching bisim.)	83,414	194,271	71
$\mathsf{m}^{\mathsf{ie}}_{\mathsf{opt,fin}}$	133,864	1,699,188	71
reduced (strong/branching bisim.)	16,620	69,550	71
$\mathsf{m}^{\mathsf{req}}_{\mathsf{opt,fin}}$	41,264,499	743,604,503	230
reduced (strong bisimulation)	29,586,657	58,206,010	230
reduced (branching bisimulation)	21,643,798	50,263,151	230
$\mathsf{m}^{\mathsf{req,ie}}_{\mathsf{opt,fin}}$	44,643,962	1,538,273,570	245
reduced (strong bisimulation)	4,371,205	12,321,042	245
reduced (branching bisimulation)	3,135,659	11,085,496	245
$\mathsf{m}^{\mathsf{q}}_{\mathsf{opt,fin}}$	$>$ 467,940,404	$>$ 2,937,662,934	?
$\mathsf{m}^{\mathsf{q,ie}}_{\mathsf{opt,fin}}$	$>$ 409,969,247	$>$ 2,726,093,658	?

Table 7.4: Size of state space of finite version of our models, before and after reduction. (Only incomplete numbers for models $\mathsf{m}^{\mathsf{q}}_{\mathsf{opt,fin}}$ and $\mathsf{m}^{\mathsf{q,ie}}_{\mathsf{opt,fin}}$ available—state space generation for them aborted, probably because the state space generator ran out of memory.)

space, and therefore models have to be finite. We achieved this by restricting the number of times that the server accepts a new client connection to a finite number, namely three. With three connections we can trigger the majority of the interesting scenarios and verify the requirements. Still, the number is low enough to allow state space generation. Table 7.4 shows the sizes of the state spaces of these model variants.

Model optimisation To make model checking feasible, we needed to optimise the model. Thus, we produced the optimised model, $\mathsf{m}_{\mathsf{opt}}$, because in the original model, $\mathsf{m}_{\mathsf{dev}}$, a single event—a client connecting to the server—could result (non-deterministically) in multiple different configurations of the client administration data structures. This badly affected state space generation: it took several hours, whereas with model $\mathsf{m}_{\mathsf{opt,fin}}$ it took in the order of minutes. As discussed in Section 7.5.6, the speed of testing with JTorX is influenced in a similar way. The optimised model $\mathsf{m}_{\mathsf{opt}}$ is almost fully deterministic, which greatly reduces the work of JTorX' on-the-fly determinization algorithm.

Model correctness We checked Requirements 2, 3, 4, 7 and 8 on model $\mathsf{m}_{\mathsf{opt}}$ and on model $\mathsf{m}^{\mathsf{req,ie}}_{\mathsf{opt}}$, using the evaluator4 tool of the CADP tool set. We found that these requirements are all satisfied by the model.

To investigate feasibility of checking the other requirements on model $\mathsf{m}^{\mathsf{req,ie}}_{\mathsf{opt}}$, we also checked requirements 9 and 10 on it, and checked requirement 7 with messages that contain a list of two services. Listing 2 shows (some) of the properties that we used to check requirement 2.

```
1 (* each ConnectRequest is followed by a ConnectAcknowledge *)
2 [ true* . ConnectRequest ]  < { ConnectAcknowledge ?m:Nat } > true
3
4 (* each ConnectAcknowledge for a connection m
5    is followed by a corresponding ConnectAuthenticate *)
6 [ true* . { ConnectAcknowledge ?m:Nat } ]
7 < { ConnectAuthenticate !m } > true
8
9 (* if, after sending a ConnectAcknowledge for connection m,
10   the server does not receive a corresponding ConnectAuthenticate,
11   it will not send any other message on the connection  *)
12 [ true* .  { ConnectAcknowledge ?m:Nat } ]
13 [ (not  { ConnectAuthenticate !m })* .
14 ( { ServiceAdvertisementEvent !m ?n:Nat }
15 | { ServiceEnquiryEvent !m ?n:Nat }
16 | { Subscribe !m !"mConnectEvent" }
17 | { Subscribe !m !"mDisconnectEvent" }
18 | { Unsubscribe !m !"mConnectEvent" }
19 | { UnsSubscribe !m !"mDisconnectEvent" }
20 )
21 ] false
```

Listing 2: MCL formulas—input for model checker evaluator4—used to verify Requirement 2.

For those requirements that we checked, we typically formulated and checked multiple formulas, to verify a single requirement. For example, for requirement 9 we not only tried to verify that the intended destination receives the private message that is sent to it, but also that a client c only receives a private message when there was a client that sent that message with c as destination.

7.5 Model-Based Testing of the XBus

This section describes the model-based testing activities from Section 7.3.5 in more detail. We focus on (1) test architecture, (2) faults discovered, and (3) test coverage.

7.5.1 Model-based integration testing in the first phase

Test architecture We used the test architecture from Fig. 7.6 on the following page to test implementation i_1. We wanted to test the XBusManager, but we could not access it directly. We accessed it via a *test context*: everything between the adapter and the XBusManager. We provide stimuli to the XBusManager using three instances of XBusGenericClient (c in Fig. 7.6), each of which is connected to the XBus via its own TCP connection. We observe the responses from the XBus not via the XBusGenericClient, but via a direct (testing) interface that has been added to XBus—t in Fig. 7.6. This interface is provided by the JTorXTestableXBusManager in the Engine package, see Fig. 7.2 on page 224. JTorXTestableXBusManager overrides the function that XBus uses to send a message to a specified client: instead, it logs the message name and relevant

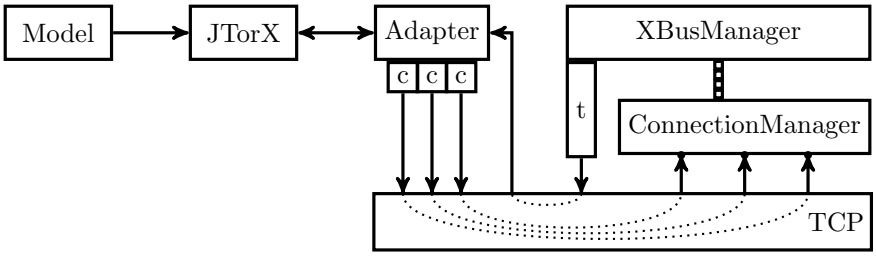


Figure 7.6: Test Architecture used in the first phase: JTorX provides stimuli to XBus via generic clients (c), and observes responses via test interface (t), both over TCP. Disadvantage: we have extended XBus with test interface t. Advantage: we do not have to extend the model with FIFO queues to deal with possible reordering of XBus responses by TCP.

parameters in the textual format that JTorX expects. Additional glue code—the adapter—provides the connection between JTorX and the XBusGenericClient instances on the one hand, and between JTorX and test interface t on the other hand. From JTorX, the adapter receives requests to apply stimuli, and from test interface t, it receives observed responses. The adapter forwards the received responses to JTorX without additional processing. For each received request to apply a stimulus, the adapter uses XBusGenericClient methods to construct a corresponding XBusMessage message, and send it to the XBus server (except for the $\mathsf{Conn}_{\mathsf{req}}$ message, for which XBusGenericClient only has to open a connection to XBus).

The adapter is implemented as a C# program that uses the Client package (see Fig. 7.2) to create the three XBusGenericClient instances, which in turn use the Communications package to interact with the XBus. The main functionality implemented in the adapter is the mapping between XBus messages and the corresponding XBusGenericClient methods, and the corresponding XBusGenericClient instances. Due to the one-to-one mapping that exists between these—by design, recall Section 7.3.2— implementing this mapping was rather straightforward.

Also JTorX and the adapter communicate via TCP: the adapter works as a simple TCP server to which JTorX connects as a TCP client.

Although it may seem that the Communications package does not play a role during model-based testing with this test architecture—also because we mentioned that we excluded it from the model—this package certainly is tested during model-based testing, as follows. The Communications package is used normally in the XBus to receive the messages that clients send to it. Moreover, the one functionality of the Communications package that is not used in the XBus itself in this test architecture—the functionality to send messages over TCP—is used by the XBusGenericClient instances that are used to send the stimuli to the XBus.

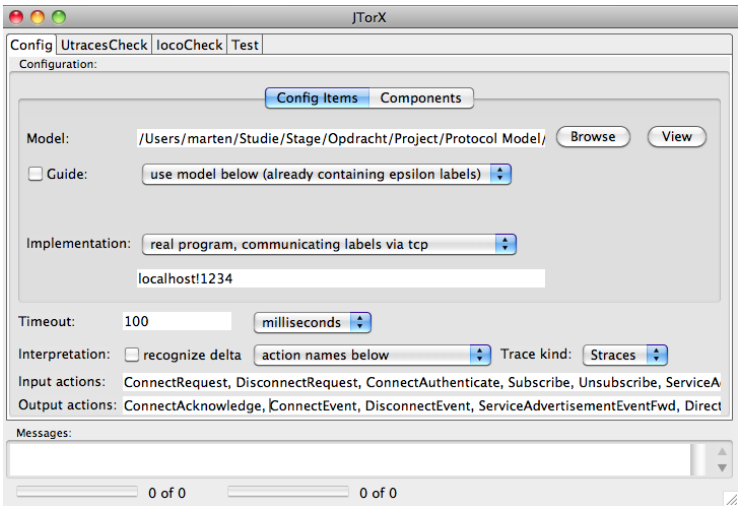


Figure 7.7: Screen shot of the configuration pane of JTorX, set up to test XBus. JTorX will connect to (the adapter that provides access to) the system under test via TCP on the local machine, at port 1234. The bottom two input fields list the input and output messages.

Preserving observation order As we wrote in Section 7.3.5, the TCP connections between XBus server and its client may reorder concurrently sent messages. We also wrote that we assumed that this would not be a problem for stimuli. For observations we added an interface, t in Fig. 7.6, to allow JTorX to observe responses in the order in which they were created. For each incoming message the XBusManager sends at most one response to each connected client (during the analysis phase, requirement 14 was added to make sure that this property remained valid, even when we extended the model with local messages), and the order in which the XBusManager sends the responses is exactly reflected in model $\mathsf{m}_{\mathsf{dev}}$. As we mention in Section 7.5.2, in the analysis phase we also looked at other ways to deal with this issue, e.g. by extending the model with a queue context, hence models $\mathsf{m}^{\mathsf{q}}_{\mathsf{opt}}$ and $\mathsf{m}^{\mathsf{q,ie}}_{\mathsf{opt}}$. However, we found that code coverage obtained with these models was identical to code coverage obtained with the corresponding models without queues, but the test runs took a lot (5 to 10 times) longer.

Running JTorX Once we had the model ($\mathsf{m}_{\mathsf{dev}}$), the XBus implementation to test (i_1), and the means to connect JTorX to it, testing was started. We ran JTorX in random mode. In the first phase, we used JTorX via its graphical user interface. Figure 7.7 shows the settings in the JTorX GUI. These include the location of the model file, the way in which the adapter and the XBus are accessed, and an indication of which messages are input (from the XBus server perspective) and which ones are output.

Bugs found in the first phase One of the most interesting parts of testing is finding bugs. In this case, not only because it allows improving the software, but also because finding bugs can be seen as an indication that model based testing is actually helping us. We found 5 bugs when testing implementation i_1 (and a few more when testing implementation i_2—these we discuss in Section 7.5.2). Typically these bugs were found within 5 minutes after the start of a test. Some of them are quite subtle:

1. The $\mathsf{Notif}_{\mathsf{disc}}$ message was sent to unsubscribed clients. This was due to an if-statement that had a wrong branching expression.
2. The $\mathsf{Serv}_{\mathsf{ann}}$ message was sent (also) to unauthorised clients. Clients that were still in the handshake process with the server, and thus not fully authenticated, received the $\mathsf{Serv}_{\mathsf{ann}}$ message. To trigger this bug one client has to (connect and) announce its service while another client is still connecting.
3. The message subscription administration did not behave correctly: a client could subscribe to one item, but *not* to two or more. This was due to a bug in the operation that added the subscription to the list of a client.
4. The same bug also occurred with the list of provided services. It was implemented in the same way as the message subscription administration.
5. There was a flaw in the method that handles Unsub messages. The code that extracts subscriptions from these messages (to be able to remove them from the list of subscriptions of the corresponding client) contained a typing error: two terms in an expression were interchanged.

All these bugs concern the order in which protocol messages must occur. Therefore, it is our firm belief that they are much harder to discover with unit testing.

7.5.2 Model-based testing in the second phase

In the second phase, we ran JTorX on implementation i_2, with all (infinite) models except $\mathsf{m}_{\mathsf{dev}}$. We did not test i_2 against model $\mathsf{m}_{\mathsf{dev}}$, because $\mathsf{m}_{\mathsf{dev}}$ was designed for the test architecture of the first phase. Model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ is a version of $\mathsf{m}_{\mathsf{dev}}$ that exactly catered for the test architecture used in the second phase.

Test architecture In the second phase, we chose a different architecture, see Fig. 7.8. Rather than the—quite complex—set-up from the first phase, we chose to observe the SUT's responses via the same connections that are also used for the stimuli. This led to a simpler adapter and test set-up, but required a more complex model: $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$. Observations are no longer observed via the test interface (block t in Fig. 7.6), but they were sent through TCP. For each XBus client—recall that JTorX plays the role of three XBus clients—there was a separate TCP connection between XBus server and adapter, such that responses might arrive at the adapter in an order that differed from the order in which they were sent. We adapted the model to reflect this, in two ways. In model

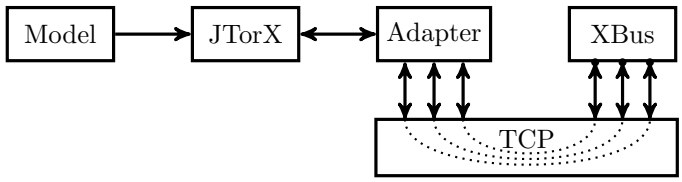


Figure 7.8: Test architecture used in the second phase: JTorX connects to XBus over TCP, where the TCP connections (one connection for each client of which JTorX plays the role) are used for both stimuli and responses. Advantage: XBus is unchanged. Disadvantage: we do have to extend the model to deal with the possibility that TCP reorders concurrently sent XBus responses (the responses sent for a single incoming message).

$\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ we directly included the different orders in the model. In models $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie2}}$ we extended the model with a queue model that describes the behaviour of the TCP channel.

Running JTorX In the second phase, we mostly invoked JTorX via its non-graphical interface—a recent development, which did not yet exist in the first phase. We ran the tests on the same machine that we also used for the model-checking—it has two quad-core Intel Xeon X5555 processors and 144 GB of memory[3]. To ensure that the Java virtual machine that ran JTorX had ample memory, we invoked it with command line options that allowed it to use 8GB.

We tested implementation i_2 with all models from Figure 7.5, except for $\mathsf{m}_{\mathsf{dev}}$ which required the test architecture from the first phase. Table 7.5 on page 243 shows test execution time for runs of 10,000 and 250,000 test steps, and maximal attainable code coverage.

We discuss coverage results and test execution time in Sections 7.5.3–7.5.6.

Bugs found in the second phase Testing in the second phase revealed bugs in implementation i_2. This does not help to improve the quality of i_1—i_2 was developed separately from i_1, based on textual information about i_1—but it does demonstrate the ability to find errors with our approach. We mention two bugs that we found most illustrative.

1. Implementation i_2 contained a race. Its TCP listener, while waiting for new connections, would, after accepting a new connection c, do the following:

   (a) first obtain a data structure for the connection information, then

   (b) send a message m to the dispatcher, to inform it of c,

[2]Note that $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ was derived from $\mathsf{m}_{\mathsf{opt}}$, rather than from $\mathsf{m}_{\mathsf{dev}}$, mainly because $\mathsf{m}_{\mathsf{opt}}$ was smaller and therefore easier to adapt.

[3] For model-based testing that machine was quite a bit oversized: we have also done test runs on a Macbook with a 2.4GHz Intel Core 2 Duo processor with 8GB of memory.

(c) finally, update the data structure with details necessary to send messages over the new connection.

This sequence contains a race: the implementation breaks when the dispatcher, after receiving message m, tries to send a $\mathsf{Conn}_{\mathsf{ack}}$ to the client, before the data structure update—necessary to be able to send that $\mathsf{Conn}_{\mathsf{ack}}$—has taken place. We could trigger this error with each of our models.

2. The adapter contained a resource leak. During a test run, it created and closed many connections, but when closing a connection it did not release all associated resources. We found this when a long test run failed after approximately 125,000 test steps. Note that MBT excels at long test runs. This bug can easily be found by any test that runs long enough.

While extending model and implementation, we occasionally tested on purpose with old versions of the model or implementation, to see whether the resulting inconsistencies were found. Again, we mention two examples.

1. The list of services that appears in a $\mathsf{Serv}_{\mathsf{rsp}}$ message was not sorted correctly. We initially—on purpose—omitted code to do this sorting from the implementation, to see whether this bug would be detected; it was.

2. One version of the model incorrectly prescribed that in response to a $\mathsf{Local}_{\mathsf{req}}$, first a $\mathsf{Local}_{\mathsf{ind}}$ message is sent to the destination, and only then $\mathsf{Notif}_{\mathsf{local}}$ messages are sent to the subscribers. In the implementation, $\mathsf{Local}_{\mathsf{req}}$ messages are handled in precisely the same way. However, in a test run, a $\mathsf{Notif}_{\mathsf{local}}$ was observed first (due to reordering of responses by the TCP test context), while a $\mathsf{Local}_{\mathsf{ind}}$ was expected first. We adapted the model to allow observation of $\mathsf{Notif}_{\mathsf{local}}$ and $\mathsf{Local}_{\mathsf{ind}}$ in arbitrary order.

Once models and implementation i_2 were stable, we ran tests of up to 250,000 test steps without finding further errors.

7.5.3 Model coverage

LPS summand coverage We used LPS summand coverage as our model coverage metric, i.e., the percentage of LPS summands that were hit during test execution. To test with an mCRL2 model, it has to be translated (by mcrl22lps from the mCRL2 toolset) into an intermediate format called *Linear Process Specification (LPS)*. JTorX then accesses such LPS via tool lps2torx, also from the mCRL2 tool set[4]. An LPS represents a set of nondeterministic alternatives, called *summands*. A summand is a syntactic expression over model variables and parameters, containing a guard, an action to be executed, and a recursive

[4] The LTSmin tool set also contains a tool lps2torx, that JTorX also can use to access an LPS. Throughout the experiments described in this chapter we used $\mathsf{lps2torx}_{\mathsf{mCRL2}}$, except in the analysis of testing time, where, as discussed in Section 7.5.6 on page 243, we also used $\mathsf{lps2torx}_{\mathsf{LTSmin}}$. As we did above, when needed, we use subscripts to distinguish between these two lps2torx instances.

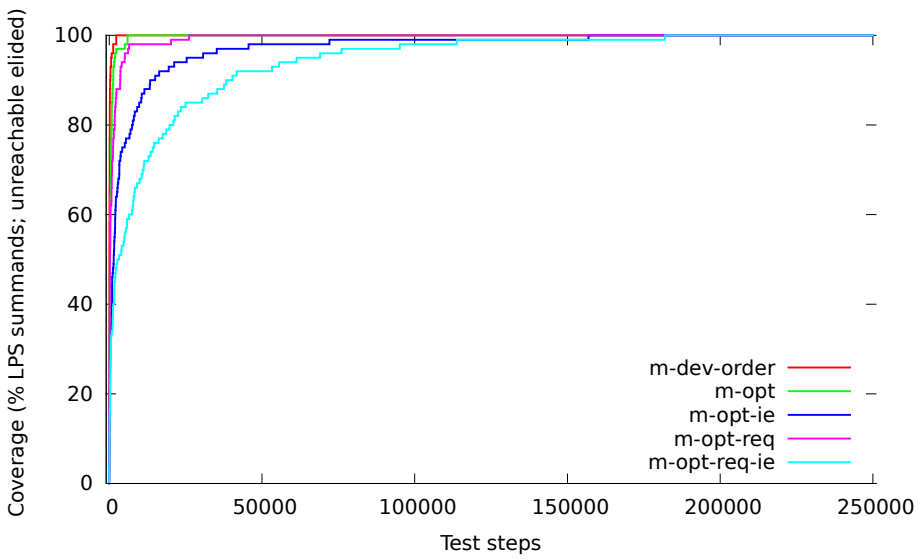


Figure 7.9: Model coverage obtained in test runs of 250,000 test steps.

invocation of the process. They express, respectively, when this alternative is enabled, the action to be taken, and the next state.

To measure LPS summand coverage, we extended the lps2torx tool from the mCRL2 tool set, so that each summand is assigned a unique identifier[5]. During test execution, we record the identifiers of all executed summands, and thus LPS summand coverage can easily be computed.

Just as programs may contain unreachable code, models may contain unreachable summands, i.e. summands which are never executed because their guard can never be enabled. We do not take unreachable summands into account when we compute model coverage; we used model checking to do the analysis of summand reachability.

Coverage results Figure 7.9 and Fig. 7.10 show the model coverage for test runs of 250,000 resp. 10,000 steps on models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, $\mathsf{m}_{\mathsf{opt}}$, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$, and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$; recall that testing is done by taking random test steps. We see that models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ and $\mathsf{m}_{\mathsf{opt}}$ reach 100% code coverage quickly (within 6,000 steps), model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$ takes somewhat more steps (slightly over 26,000), while model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$ needs about 150,000 steps, and model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ needs about 180,000 steps. We do not show model coverage results for models $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$, because the unreachable summand analysis did not terminate.

[5]In the mean time, such functionality has been integrated in the lps2torx tool in the mCRL2 tool set.

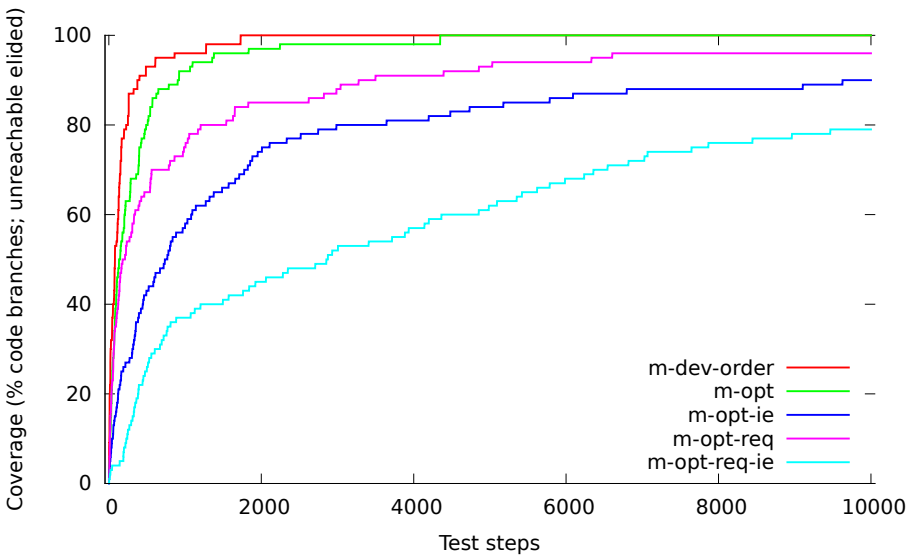


Figure 7.10: Model coverage obtained in test runs of 10,000 test steps.

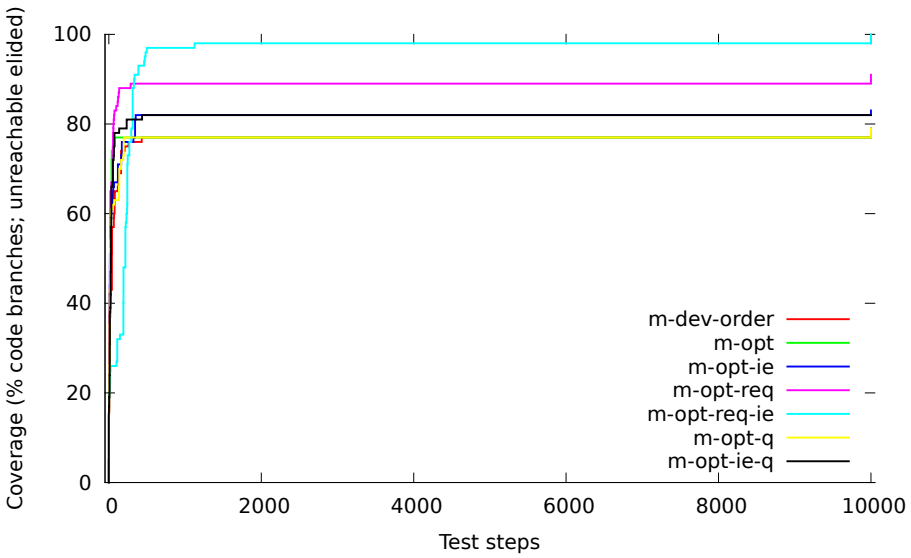


Figure 7.11: Code coverage (on i_2) obtained in test runs of 10,000 test steps. Note that coverage for m_{dev}^{order}, m_{opt} and m_{opt}^{q} converges to the same level, and so does coverage for m_{opt}^{ie} and $m_{opt}^{q,ie}$. From the test runs of 250,000 steps, we obtain an almost identical plot (not shown).

7.5.4 Code coverage

Branch coverage We used branch coverage as our code coverage metric. Branch coverage [Mye79] is a standard code coverage metric that counts the percentage of branches traversed in a program's control flow graph during test execution. It was measured by instrumenting the code.

Initial coverage analysis showed that 19 blocks were unreachable, because of the following reasons. Two blocks handle operating system errors, which never occurred in our case—testing operating system related functionality requires a different test set-up, where we simulate the operating systems, and deliberately insert errors. Four blocks handle Sub and Unsub messages with an invalid message type—such messages do not appear in our most complete model, though they can easily be added (we leave that for future work). Finally, thirteen blocks handle inherently unreachable cases. For example, when an incoming message is being handled, the list of active connections will always contain at least one element, namely the sending connection. Therefore, the code that looks up the connection record for a given connection will never encounter an empty list of connections, and thus the code that handles the case of an empty list is unreachable. One could use static analysis tools to show that these blocks can never be executed, and then safely remove these blocks—but this falls beyond the scope of this chapter.

Since coverage should, in our opinion, measure the code covered by a specific test as a percentage of what can be covered, we left out the unreachable blocks from our coverage analysis.

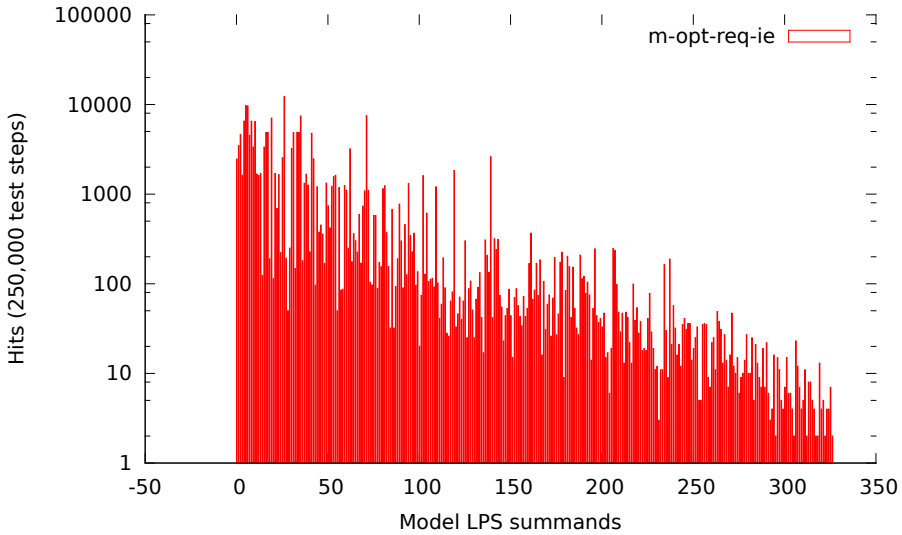


Figure 7.12: Model coverage obtained in a test run of 250,000 test steps with model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, showing, for each LPS summand of the model, how often it is hit. LPS summands are ordered, in order of first "hit".

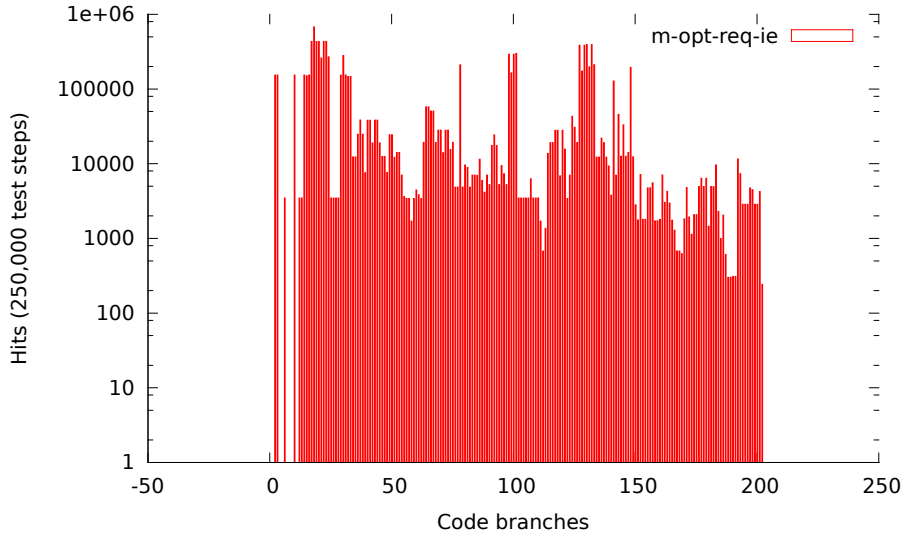


Figure 7.13: Code coverage obtained in a test run of 10,000 test steps with model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, showing, for each branch of the implementation, how often it is hit. Branches are ordered, in order of first "hit".

Coverage results Figure 7.11 shows the code coverage results for all models, for test runs of 10,000 steps. These experiments reveal two interesting phenomena: (1) the maximum attainable code coverage varies per model, and (2) the use of queues does not affect maximum attainable coverage.

The maximum attainable code coverage figures are shown in Table 7.5 on the facing page. As expected, we see that, the more complete a model is, the higher the maximum code coverage is: $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ is the least complete model with 79% maximal code coverage: since $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ does not contain self-defined XBus messages, it can not trigger all behaviour in the implementation. Model $\mathsf{m}_{\mathsf{opt}}$ reached the same coverage. This is no surprise either, because $\mathsf{m}_{\mathsf{opt}}$ is an optimised version of $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$.

As we explain in Section 7.5.6, test execution from $\mathsf{m}_{\mathsf{opt}}$ is significantly faster. Also $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$ reaches 79% code coverage. This is interesting, because, apparently, the use of queues does not affect maximal code coverage. Indeed, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$ and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$ reach the same maximal coverage, namely 83%. The most complete model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ reaches 100% coverage. From these experiments, we show that measuring code coverage is important: if 100% code coverage cannot be reached, then the model is incomplete, so not all behaviour can be tested. If this is the case, we advise to extend the model.

7.5.5 Distribution of coverage

In Figures 7.12 and 7.13 we see that all "hit" code blocks and all "hit" LPS summands were hit multiple times, although the number of hits is not evenly

model	*10,000 steps*	*250,000 steps*	*250,000 steps jittyc*	*max att. code coverage*
$\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$	18 minutes	24 hours	15.75 hours	79%
$\mathsf{m}_{\mathsf{opt}}$	5 minutes	2 hours	2 hours	79%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{ie}}$	6 minutes	2.5 hours	2.25 hours	83%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req}}$	6 minutes	3 hours	2.5 hours	91%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$	7 minutes	3 hours	2.5 hours	100%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{q}}$	37 minutes	77.75 hours	69.75 hours	79%
$\mathsf{m}_{\mathsf{opt}}^{\mathsf{q,ie}}$	30 minutes	44.5 hours	38 hours	83%

Table 7.5: Wall-clock time for runs on implementation i_2, using JTorX in non-GUI mode, and the maximal attainable code coverage for each model.

distributed. (Note the logarithmic scale on the vertical axis.) For the code coverage, obviously, certain blocks are hit quite often, e.g. because they are hit whenever an incoming message has to be processed, whereas other blocks are only hit once, during initialisation—this explains the "gap" slightly at the right of block "0" in Figure 7.13.

For the plot of the model coverage (Fig. 7.12), it could be interesting to separate the stimuli from the responses, to see to what extent the following hypothesis is true: for stimuli, there is a direct correspondence between the number of actions that are generated from a summand, and the number of times that the summand is "hit".

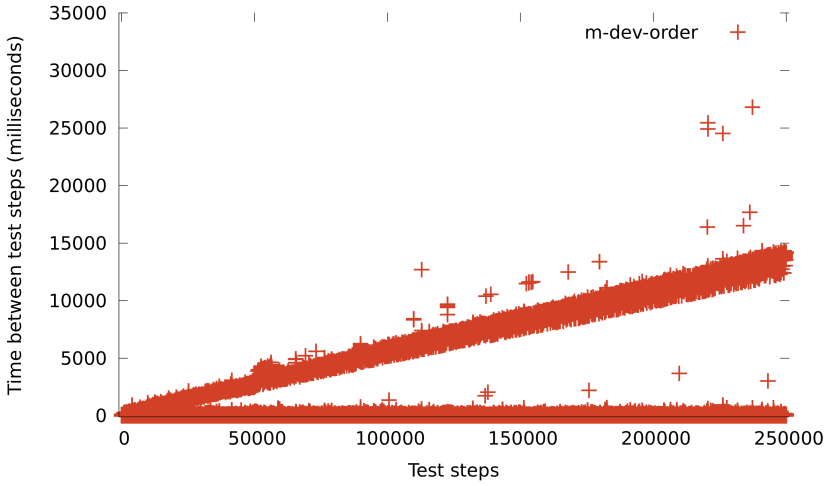
7.5.6 Testing time

We also analysed the test execution times, see Table 7.5 and Figures 7.14–7.16.

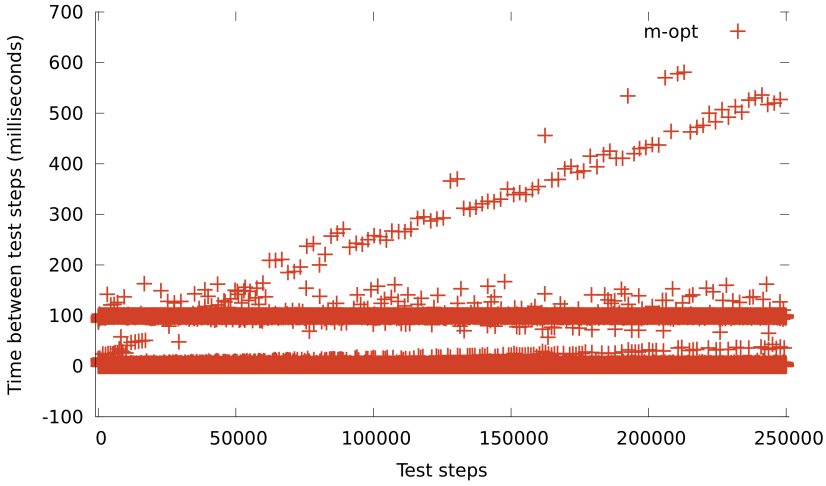
Table 7.5 shows the test execution times for the runs of 10,000 and 250,000 test steps. The 4^{th} column shows the effect of enabling option *jittyc* of tool lps2torx; all time-related results that we show in the plots were obtained with this option enabled. When option *jittyc* is enabled, lps2torx uses a jit-compiled rewriting engine, instead of its interpreting rewriting engine—the more rewriting that has to be done, the greater the gain. Jit-compilation takes approx. 11 seconds at the start of a test run; this is not shown in Figures 7.14b, 7.15a and 7.15b, to avoid compressing the scale on the vertical axis.

Figures 7.14a, 7.14b, 7.15a and 7.15b present scatter plots showing, for each test step generated from respectively model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, $\mathsf{m}_{\mathsf{opt}}$, $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ (accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$), and model $\mathsf{m}_{\mathsf{opt}}$ (accessed using $\mathsf{lps2torx}_{\mathsf{LTSmin}}$), the amount of time in milliseconds it takes to execute. ($\mathsf{lps2torx}_{\mathsf{mCRL2}}$ and $\mathsf{lps2torx}_{\mathsf{LTSmin}}$ were introduced in the footnote on page 238.) Thus, in these plots a point at test step 12743 at testing time 300, means that the 12743^{th} test step took 300 *ms*. Figures 7.16a and 7.16b present the same information differently: for each test step duration d, they shows the number of test steps that took d *ms* to execute.

Figure 7.14b shows two tick areas. One is below 20 *ms*, showing that most test steps took less than 20 *ms*. Another tick area is around 100 *ms*, which is exactly the value of the quiescence timer. This is to be expected: if one wants to observe quiescence (i.e. absence of outputs), one observes the system for (in our case) 100 *ms* and sees if any outputs are produced. Thus, if quiescence is observed, this step takes exactly 100 *ms*. Also the plots for models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ and

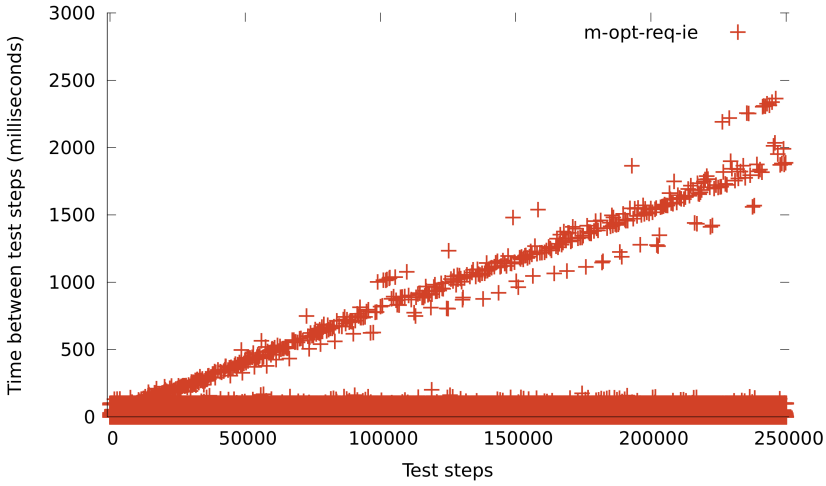


(a)

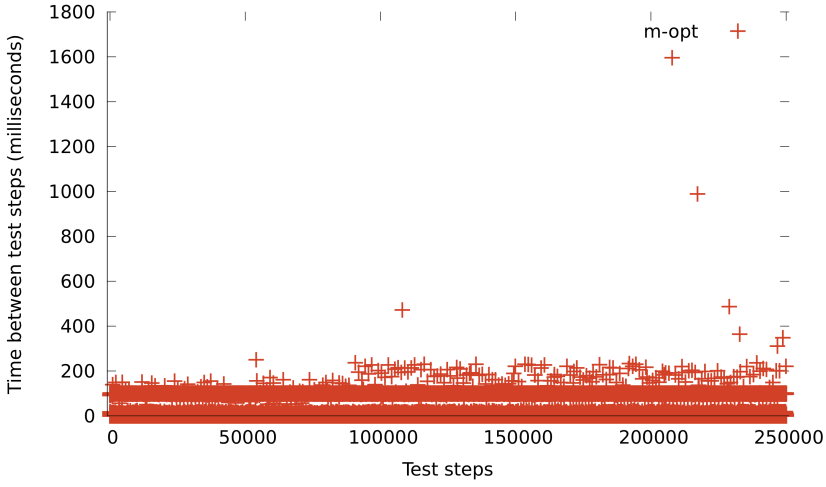


(b)

Figure 7.14: Time (*ms*) between test steps in run of 250,000 test steps with models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ (a) and model $\mathsf{m}_{\mathsf{opt}}$ (b), accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$.

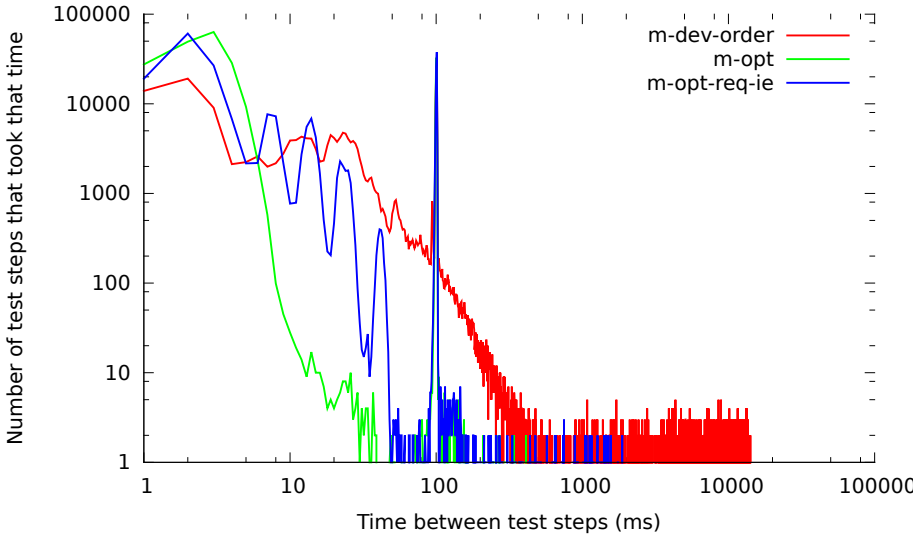


(a)

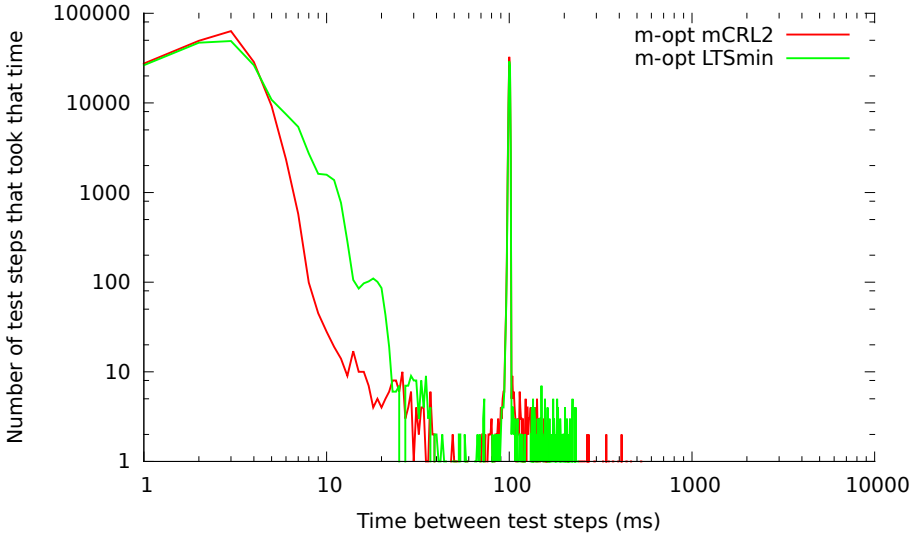


(b)

Figure 7.15: Time (*ms*) between test steps in run of 250,000 test steps with model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$ (a), and with model $\mathsf{m}_{\mathsf{opt}}$, accessed using $\mathsf{lps2torx}_{\mathsf{LTSmin}}$ (b).



(a)



(b)

Figure 7.16: Distribution of the time spent per test step, for runs of 250,000 steps, with models $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$, $\mathsf{m}_{\mathsf{opt}}$, and $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$, accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$ (a), resp. with model $\mathsf{m}_{\mathsf{opt}}$, accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$ resp. $\mathsf{lps2torx}_{\mathsf{LTSmin}}$ (b). Note logarithmic scale on both axes in both figures.

$\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ contain these same two tick areas, but we have to zoom in sufficiently to distinguish them; they are also visible separately in Figures 7.15a and 7.15b but not in Fig. 7.14a, mainly due to the different scale on the vertical axis.

Figures 7.14a–7.15b show the time needed per test step as a function of the total number of test steps executed. Those plots where the model was accessed using $\mathsf{lps2torx}_{\mathsf{mCRL2}}$ show few (Fig. 7.14b) resp. a significant portion (Fig. 7.14a, 7.15a) of test steps whose execution time grows linearly with the number of test steps executed. This may be surprising, because one would assume that executing a single test step requires a fixed amount of time. We attribute the linear behaviour to the growth of state mappings in $\mathsf{lps2torx}_{\mathsf{mCRL2}}$: both lps2torx instances maintain a mapping between the state representation that they uses internally, and the state identifiers (numbers) that they exchange over their interface with JTorX. With $\mathsf{lps2torx}_{\mathsf{mCRL2}}$, more or less regularly, a map insertion takes more time, when the map adjusts itself to cope with the ever growing number of entries. With $\mathsf{lps2torx}_{\mathsf{LTSmin}}$ this appears to happen much less often, but when it does happen, it takes much longer than with $\mathsf{lps2torx}_{\mathsf{mCRL2}}$. (We did not do a full performance comparison between $\mathsf{lps2torx}_{\mathsf{mCRL2}}$ and $\mathsf{lps2torx}_{\mathsf{LTSmin}}$, we leave that for future research. The measurements that we did suggest that $\mathsf{lps2torx}_{\mathsf{LTSmin}}$ spends, in general, slightly more time on a model-access request from JTorX than $\mathsf{lps2torx}_{\mathsf{mCRL2}}$. Nevertheless, with both lps2torx instances, the same test run of 250,000 test steps with model $\mathsf{m}_{\mathsf{opt}}$ took the same two hours of wall clock time.) The difference in severity of the linear growth, that we see across Figures 7.14a–7.15a, we attribute to the different numbers of states that the mappings contain. After 250,000 test steps, with model $\mathsf{m}_{\mathsf{opt}}$ the mapping only contains approx. 900,000 states; with model $\mathsf{m}_{\mathsf{opt}}^{\mathsf{req,ie}}$ approx. 6,300,000, and with model $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$ approx. 35,000,000. To understand this huge difference, recall that in model $\mathsf{m}_{\mathsf{dev}}$ (and thus also in $\mathsf{m}_{\mathsf{dev}}^{\mathsf{order}}$) a single event could result (non-deterministically) in multiple different configurations of the client administration data structures, i.e. in multiple different states, whereas $\mathsf{m}_{\mathsf{opt}}$ is almost fully deterministic. Note however, that in Figures 7.14a–7.15b the long test steps are a small fraction of all steps. Figures 7.16a and 7.16b show that the majority of all test steps take less than 1000 *ms*. In our experiments, test step derivation time was not a bottleneck, but it could be an issue when testing real-time systems.

7.6 Findings and Lessons Learned

7.6.1 First phase

The internship in a time perspective So how long did it take to create the artefacts for model-based testing, namely the model, the test interface and the adapter? Programming and simulating the model took 2 weeks, or 80 hours. The test interface was created in a few hours, since it was designed to be loosely coupled to the engine. It was a matter of a few dozens lines of code. The adapter was created in two days, or 16 hours. Thus, given the total project time of 14 weeks, creating the artefacts needed for model-based testing took thus about

V-model phase	Time
1. XBus Requirements	7%
2. XBus Design	14 %
3. Implementation and 4. Unit Testing	60 %
5. Integration Testing (model-based, incl adaptors)	17%
6. Acceptance Testing	2%

Table 7.6: Estimation of the time spent in the first phase, in terms of the activities of the V-model of Fig. 7.1.

17% of our time. Table 7.6 shows an estimation of how the time was spent.

The modelling process Writing a model takes a significant amount of time, but also forces the developer to think about the system behaviour thoroughly. Moreover, we found it extremely helpful to use simulation to step through the protocol, before implementing anything. Making and simulating a model gives a deep understanding of the system, in an early stage of development, from which the architectural design profits.

7.6.2 Second phase

After a thorough analysis of the model-based testing process that was carried out in the first phase, the question remains how good the approach was. We reflect on the questions raised in Section 7.1.2.

How good was the model? Model $\mathsf{m}_{\mathsf{dev}}$ did its job, but there is certainly room for improvement. In particular, completeness with respect to the requirements and bad weather behaviour could be improved.

Was the testing thorough enough? Given model $\mathsf{m}_{\mathsf{dev}}$, we believe that testing was thorough enough, in the sense that the testing time sufficed to fully cover model $\mathsf{m}_{\mathsf{dev}}$, and to cover as much of the implementation code as was possible with model $\mathsf{m}_{\mathsf{dev}}$. On the other hand, model-based testing is as good as the model is, so more complete models also mean better testing, resulting in higher code coverage.

What can we say about code coverage? The model $\mathsf{m}_{\mathsf{dev}}$ does not reach 100% code coverage. For that, more complete models are required.

Would model checking have helped to produce better code? Formalising the requirements, which is a prerequisite for model checking, helps to improve the model, and therefore the code. However, model checking requires great effort, because models need to be made finite and efficient. For model-based testing, this was not needed, since performance was not an issue here.

Despite these observations, we still believe in our approach during the first phase. If we had to redo the XBus development we would take a very similar approach, but (1) invest more effort in the modelling phase: trace back the requirements, and make models input-complete, and (2) measure coverage.

7.7 Conclusions and Future Research

We conclude that the approach of using formal methods in both the design step and the integration testing step of the V-model was a success: with a relatively limited effort, we found five subtle bugs. We needed 17% of the time to develop the artefacts needed for model-based testing, and given the errors found, we consider that time well spent. Moreover, for future versions of the XBus, JTorX can be used for automatic regression tests: by adapting the mCRL2 model to new functionality, one can detect automatically if new bugs are introduced.

Our post-case study analysis showed that a 14 week development process is feasible but short: the model quality would have benefited from more attention—in particular, tracing the requirements would have been helpful.

The test execution time analysis results suggest that performance improvements can be made by optimising the interface between JTorX and the lps2torx tools of the mCRL2 and LTSmin tool sets; for this, further measurements and analysis will be necessary.

Thus, the post-internship analysis gave us a deeper understanding of the limitations and the successes of the work done during the internship, an increased understanding of what factors are responsible for the successes, and valuable feedback that may help us to improve our tools.

Chapter 8

Evidence

In this chapter we present the evidence that we use to validate the design requirements that we stated in Chapter 1. We focus on the non-functional requirements—the functional ones have been incorporated in our design, as indicated in the discussion of each of its elements; Section 1.2.3 contains an overview of where each requirement is discussed. TorX and JTorX, the two implementations of our design (see Appendix A), together referred to as (J)TorX, have been used for teaching and for case studies. We first give an overview of the educational and industrial use of (J)TorX that we are aware of—already several years ago we made both tools available for free download, without restriction on use, an without obligation to inform us of interesting use, so this overview will be incomplete. We then present responses to a questionnaire about (J)TorX. We conclude this chapter with an evaluation of the design requirements w.r.t. the evidence that we presented.

Remainder of this chapter In Section 8.1 we give an overview of case studies carried out with (J)TorX; in Section 8.2 of the use in industry, in Section 8.2 of the use in research, and in Section 8.3 of the use in education. In Section 8.4 we discuss the questionnaire, and in Section 8.5 we evaluate the design requirements on the basis of the evidence.

8.1 Case Studies

Multiple case studies have been done with TorX and JTorX. In Table 8.1 we give an overview of a selection of the case studies. In the description of the case studies below, we focus on the design requirements that played a role. We describe the case studies in more detail in Appendix B.

8.1.1 Conference protocol entity

In this case study we tested the protocol entity of a chatbox system. We used TorX for on-line model-based testing using LOTOS and Promela models, and

nr	*case study*	*requirements covered*	*size*	*who*	*section*
all	All case studies	1, 2, 4, 5, 7, 8, 19–21			
0	Software bus	6, 12, 17, 18, 23, 24	0	S/R	Ch. 7
1	Conference protocol entity	3, 6, 15, 22, 23	o	R	B.1
2	Easylink	6, 23	0	R	B.2
3	Highway tolling system PB	11, 24	0	R	B.3
4	Storm surge barrier EMCS	15, 19, 23, 24	0	R	B.4
5	Myrianed protocol entity	10, 23, 24	O	R	B.5
6	Rivercrossing puzzle	10, 12–16, 18	.	S	B.6

Table 8.1: Overview of selected case studies, executed by **R**esearchers and **S**tudents, with relative size ranging from '.' (tiny) to 'O' (large).

to execute tests that had been derived with the TGV tool (Req 3). The LOTOS model contained unbounded FIFO queues, which meant that it had an unbounded state space (Req 6). We also used TorX to compare the LOTOS model with an mCRL2 model of the same system, and found an error in the mCRL2 model (Req 15). We used TorX to identify correct and incorrect implementations from a set of 28 protocol entity implementations; TorX correctly identified the incorrect implementations (Req 22). TorX performed sufficiently well to be able to do this case study (Req 23).

8.1.2 Easylink

In this case study we tested the implementation of a protocol that allows television sets and other audio-visual programs to communicate over SCART, in particular, the functionality to communicate presets (channel name, frequency). We used TorX for on-line model-based testing from LOTOS and Promela models. The main challenge in this case study was that, at the start of each test run, the initial state (regarding presets) of the television set was unknown, i.e. for any stimulus at that moment, we had a huge number (10^6) of potentially valid responses (Req 6). Test derivation from LOTOS was too slow, but from Promela it was o.k. (Req 23). We explain this difference as follows. For Promela, parameterised labels were used, and for LOTOS 'normal' (instantiated) labels. Thus, for the initial state, the number of parameterised labels, enumerated from Promela, was much smaller than the number of corresponding 'normal' labels, enumerated from LOTOS. To better use our available computational resources, we used a distributed test set-up, where test derivation took place on one computer, and the Adapter ran on another one, connected via TCP telnet.

8.1.3 Highway Tolling System

In this case study we tested the payment box of a highway-tolling system. We used TorX for on-line model-based testing from LOTOS and Promela models. All communication between the payment box and its environment was encrypted with a key that we did not have; we worked around this by extending a program that had been used for traditional testing of the payment box, and that had the necessary key, with a "remote control" interface, such that it could be controlled

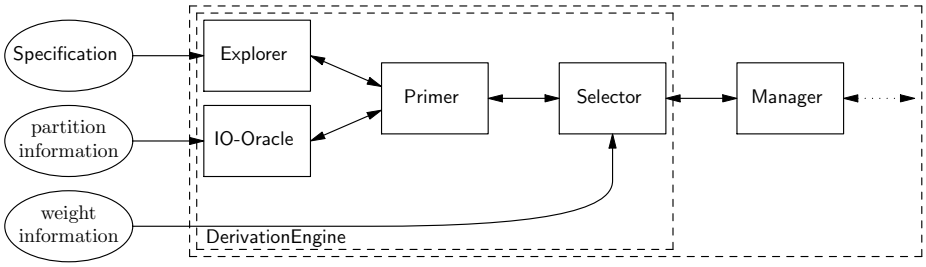


Figure 8.1: Introduction of Selector in our decomposition of Fig. 4.4.

by the Adapter (Req 24).

The model contained two kinds of stimuli: those that are part of the normal behaviour, and those that are related to error behaviour. We wanted to give priority to normal behaviour during testing, without completely ignoring the error behaviour. Therefore, we slightly adapted our tool architecture, and introduced a new component. In the architecture discussed in Chapter 3, when a stimulus has to be chosen for the next test step, it is the Manager that chooses the stimulus from the set of enabled stimuli that is provided by the Primer.

For this case study, we let the Manager delegate the choice of the stimulus to the DerivationEngine, i.e. the DerivationEngine interface was extended with two functions: The Manager can use hasInputs for its decision whether to apply a stimulus or to obtain an observation, and it can use getInput to obtain a stimulus.

We added a tool component, the Selector, that associates weights to labels, see Figure 8.1. Conceptually, the Selector delegates all requests that it gets from the Manager to the Primer, except for the getInput request: for the getInput request it makes a random selection from the inputs provided by the Primer, taking the weights into account.

This extension to the DerivationEngine interface makes it easy to integrate arbitrary test selection components into our architecture (Req 11).

8.1.4 Storm Surge Barrier

In this case study we tested the control software of a storm surge barrier. We used TorX for on-line model-based testing from a Promela model. In this case study, we experimented with timed testing: the system has requirements on the timing of the outputs, but not on the inputs. We represented discrete time

DerivationEngine:	
Signature:	
hasInputs :	$P \times (L_I \cup L_U \cup \{\delta\}) \rightarrow bool$
getInput :	$P \rightarrow L_I \uplus \{\perp\}$

Table 8.2: DerivationEngine Interface signature extension to support delegation of choice of input.

as integers in the model, each unit of time representing a second (Req 19). We tested the actual code of the system, running on a PC inside a testing environment created by the developers of the system. We created an Adapter to provide a connection to this testing environment (Req 24). Performance was important in this case, because the testing environment used shared memory to provide the Adapter access to the state of the system—the observations are created from this state information. The Adapter had to regularly access this information, in time, before it would be overwritten with fresh state information. After some fiddling we got this working (Req. 23).

After the system had gone into production, we also tested whether actual execution traces, obtained from the execution log of the system, were valid, i.e. were in the model (Req 15). We did this by using such traces both as model-to-be-simulated-as-SUT and as guide.

8.1.5 Myrianed Protocol Entity

In this case study we tested the protocol stack of a wireless sensor node. We used JTorX for on-line model-based testing from a timed automata model. The testing was done using simulated time: the actual protocol stack code was run inside a testing environment, developed by the developers of the protocol stack code, and this testing environment offered control over the progress of time as "seen" by the protocol stack code. The Adapter provides access to this testing environment (Req 24). The tool had sufficient performance to be usable (Req. 23), but that was also thanks to the use of simulated time.

8.1.6 Rivercrossing puzzle

This is not a real case study, but a lab class exercise for the Testing Techniques course given at University of Twente. The system under test is a small puzzle program in which students have to "fill in the blanks". It is tested with JTorX, from a GraphML model that is made from requirements that are given (Req. 13, 18). The students install JTorX themselves (Req. 12). The model is underspecified and non-deterministic. It is underspecified, because no inputs are enabled in the states in which the puzzle program produces output. It is non-deterministic, because it allows for two distinct implementations w.r.t. how it responds to user "errors": strict, or lenient. A strict implementation displays an error message, and resets itself to the initial state. A lenient implementation displays the same error message, and then goes to the previous state, and allows the user to retry. It turned out that this combination of underspecification and non-determinism makes it necessary to use **uioco**, to avoid unintended **fail** verdicts (Req. 10). The students typically use the visualisations to see what happens during testing (Req. 14). The implementation can interact with its environment in two ways, selectable via a command line option: using model labels over standard input and output, or using a binary encoding over standard input and output. The former can be used with the JTorX built-in Adapter for toy implementations (Req. 16); for the latter the students have to "fill in the blanks" in an Adapter that connects to JTorX using TorX-adapter interface.

The students also make mutants of the model, correct and incorrect ones, and test them with JTorX to compare them (Req. 15).

8.2 Independent Use

The case studies that we described in the previous section were all done by, or in close collaboration with, the developers of TorX and JTorX.

TorX and JTorX have also been used independent from their developers, sometimes with, sometimes without, the developers being aware of it at the time. Independent use of (J)TorX suggests that usability of (J)TorX sufficed for "third parties" to be able to do whatever they wanted to do. The unfortunate side-effect, however, is that our knowledge about the third parties' experience with (J)TorX is rather limited.

Below we mention some cases of independent use of (J)TorX in industry and an research; we mention independent use for education in Section 8.3.

Use in Industry TorX and JTorX have been used in the industry, typically in internships, assignments and project-related case studies. These we list in the sections on case studies resp. on education.

Moreover, TorX has been extended with support for an additional modelling language: STATECRUNCHER. STATECRUNCHER is a language system which implements statecharts [Tho04]. At Philips Research India Bangalore first a STATECRUNCHER Explorer was created [KB02, Kop03], and once that was ready, TorX was used with STATECRUNCHER models to test various embedded software components [Tho04].

At the model-based testing company Axini [axi], initially TorX was used as "motor" inside Axini's web-based testing tool. In this tool, over time, parts of TorX were gradually replaced by tool components that Axini developed itself, until no trace of TorX was left in it.

Last, but not least, researcher J. Tretmans uses JTorX, typically with the Rivercrossing example of Section B.6, to demonstrate the principles of model-based testing in contacts with industry. To my understanding, in one case this helped to get a project at the company.

Use in Research JTorX has been used in research, by D. Farago as research vehicle to study "Lazy on-the-fly model-based testing" [Far14, Kut14], and by S. von Styp to study the combination of time and data. Unfortunately, we are not aware of publications about the latter work.

In addition, JTorX has been used in case studies in which the main focus was not on model-based testing. In case studies on machine learning, see e.g. [AKT+13], JTorX has been used to check the learned model.

8.3 Use in Education

Both TorX and JTorX have been used in education: for practical exercises in courses, for bachelor and master assignments, and for internships.

Here we list courses and assignments in which we were involved, as well as those that took place independently.

8.3.1 Use in Courses

At the University of Twente, students have been using JTorX, and, until JTorX became available, TorX, for the lab class that is part of a master course on Testing Techniques. The students use the tools to compare models, and to test an actual implementation. With TorX they used models in the textual ALDEBARAN format; with JTorX they use graphical models in GRAPHML.

At the University of Twente, students have been using JTorX in the bachelor course Verification Engineering. In this course, small teams of students design, implement and test a small system. They use model checking in the design phase, and model-based testing using JTorX in the testing phase. Typically, their models are in MCRL2.

At the Radboud University of Nijmegen, students have been using JTorX for the lab class in a master course on Testing Techniques. Until two years ago, the lecturer chose which testing tools the students should use, which included JTorX. Now, students may choose themselves which testing tool they want to use—JTorX is demonstrated by the lecturer—and still a few groups of students choose to use JTorX.

At the dutch Open Universiteit, JTorX is used in a course on Software verification and validation. Unfortunately, we do not have further information.

At the Graz University of Technology, students have been using JTorX in the course Qualitätssicherung in der Softwareentwicklung, in the years 2010–2012. We were not involved in this, and only found out about it later. Unfortunately, we do not have information about their experience with JTorX.

8.3.2 Use in Assignments and Internships

Here we list assignments and internships in which TorX and JTorX have been used for a case study, or in which an extension to TorX or JTorX was made.

Bachelor assignments

- ToLERo: ToRX-tested LEGO Robots [Sni10] (case study with JTorX, where a simple LEGO ball sorter was constructed, together with a LEGO "test harness" which allowed applying stimuli and obtaining observations. The LEGO test harness was connected to JTorX.)
- Is Javacardsign correct and secure? [Kle12] (independent case study with JTorX)
- Model-based Testing with a B Model of the EMV Standard [dAJ12] (independent case study with JTorX)
- Testing of channel based service connectors [Leu13] (independent case study with JTorX)

Master assignments

- TorX, TestFrame and the Easy Mail Machine [Spe02] (case study with TorX)
- Model-Based Testing of Network Security Protocols in Java Card Applications [Sse06] (independent case study with TorX)
- SPEX: A Simple Promela EXplorer for TorX [vY07] (extension for (J)TorX)
- SeCo - A Tool for Semantic Test Coverage [Men08] (used off-line test cases derived with TorX)
- Model Based Testing of a PLC Based Interlocking System [tH12] (case study with JTorX)

Internships

- Timed Modelling and verification of the DO/DG component: A case study for testing a real time component with TorX, using verified timed models [Sch05] (case study with TorX)
- Experiences with Formal Engineering: Model-Based Specification, Implementation and Testing of a Software Bus at Neopost [SSBM11] (case study with JTorX)
- Model Based System Testing in Practice: Report on an Internship Performed at PANalytical [Mei12] (case study with JTorX)

8.4 Questionnaire

To obtain feedback about TorX and JTorX, we asked users to fill out a questionnaire. In the questionnaire we asked about their background, and about their experiences with installing and using the tool. We obtained 13 responses:

- 8 students who used it in the lab class of the master course Testing Techniques (TT);
- 1 student who used it also in the bachelor course Verification Engineering;
- 2 students who used it for an assignment or internship,
- 3 researchers, who all used JTorX for case studies, and of whom 2 used it for teaching, and 2 used it for their PhD research.

Below we give an overview of those questions and responses that directly correspond to non-functional requirements. In Appendix C we give the complete set of questions and answers, although also there questions and answers have been edited to improve the presentation.

8.4.1 Req. 12: it should be easy to deploy the tool (install and use)

Installation We asked how hard it was to install JTorX. Responses: 'ok' (1), 'easy' (8), 'very easy' (4).

We asked whether it was necessary to install additional software to be able to install and use JTorX. One respondent (on Mac OS X) needed to install X11; one respondent (on linux, 64-bits) needed to install two 32-bit libraries to make the automata visualization work. Furthermore, 2 respondents remarked

that they had to install yEd and GraphViz to use JTorX[1] and one researcher installed CADP.

To the question for other remarks about installing TorX and JTorX, one researcher responded "Switching from TorX to JTorX made installation much more comfortable, and hence also teaching and supervision.".

Use − GUI We asked how easy it was to use the JTorX GUI. Responses: 'ok' (8), 'easy' (5).

Use − CLI We also asked how easy it was to use the JTorX command line tool. Response: 'very easy' (1). With that response came the remark that it was helpful, easy to extend, and extremely important for large experiments.

8.4.2 Req. 13: it should be easy to create a simple model (like an automaton) for use with the tool

We asked respondents which modelling languages they used, which tools they used to make models, how hard it was to make a model, and how much time it took.

Except for one respondent, all Testing Techniques students used the graphical editor yEd that produces GraphML models. The other student used a text editor to specify the models in the GraphViz input language. The students responded that model making was 'ok' (4), 'easy' (5), 'very easy' (1), and that it took from '10 minutes' up-to '1.5 hour'. Most likely, the '1.5 hour' refers to the time spent in a class session in which the initial models are made collectively.

We also asked respondents the same questions w.r.t. making models to be used as test purposes.

All 4 Testing Techniques students who used test purposes, modelled them in GraphML using yEd. They found this 'easy' (2) to 'very easy' (2), and it took them from 'no time at all, they were previous models', via 'a few minutes (they were small guides)' up-to '10 minutes'.

8.4.3 Req. 14: the tool should provide insight in the theory and algorithms that it implements, e.g. by visualisation

We asked respondents what visualization they used, how it helped them, how it could be improved, and what visualization they missed.

They responded that they used visualization of 'model' (9), 'real implementation' (1), 'simulated model as implementation' (6), 'test run' (6), 'message sequence chart' (3), 'other' (1): 'own extensions, for visualization of lazy OTF MTB'. To the question, whether visualization was helpful, 8 responded yes, and they explained: 'get insight', 'very useful to find the error', 'it made it more easy to know where I was in the specification', 'visualise which part of the model is visited. this was very helpful to me', 'you can really see what is going on',

[1] All Testing Techniques students, except one, indicated to have used yEd, so more than 2 respondents actually installed yEd.

and 'visual feedback gave first impression and helped during experiments until everything worked fine, [and] it could be automated via command line tool'.

We got only one suggestion for improvement: 'More userfriendly it should be'.

Two respondents missed something: 'monitoring'[2], and 'statistics/plots over multiple runs would be quite helpful, but that should be part of a large test or case study management tool'.

8.4.4 Req. 16: it should be simple to connect the tool to toy implementations

We asked respondents how they connected JTorX to the SUT, where multiple answers were possible, and we asked how easy it was to connect, how much time they spent, and what the biggest stumbling blocks were. Unfortunately, the latter questions were only asked once, and not separately for each "means of connection" that a respondent gave as response to the first question.

The Testing Techniques students connected toy programs in two ways—via the built-in 'toy implementation' Adapter, and via a "fill-in-the-blanks" external one. They responded that connecting was 'ok' (7), 'easy' (3), 'very easy' (1), and that it took 'Practically nothing' to '10 minutes', 'half an hour' '1.5 hour' and '2 hour'. Most likely, the 'ok' responses and all times longer than 10 minutes reflect issues encountered while "filling-in-the-blanks" in the external Adapter.

Their biggest stumbling blocks were "I didn't fully understand how it works", and "No clear documentation of the responsibilities of the different methods (in Java)".

8.4.5 Req. 24: it should be easy to connect the tool to the system under test

The responses of the researchers, and of the student that did the case study, to the question how easy it was to connect, and how much time they spent, may give an indication how easy it is to connect the tool to the system under test. Moreover, we asked how hard it was to create an Adapter, how much time that took, and how much time it took to connect the Adapter to JTorX.

The researchers responded that connecting was 'ok' to 'hard', and that it took 'from 0 (simulated model) to couple of hours, depends very much on SUT', 'all in all about a person week (a student, a developer from industry and I)', or 'about an hour (a student of mine did it and told me)'.

Creating the Adapter took the student '2 hours', and one researcher 'all in all about 3 days (most work was a facade to hide all the web service complexities from the model and core adapter)'.

The biggest stumbling blocks were 'all the small details you have to consider, mostly due to the SUT', which another researcher phrased as 'to adjust the adapter for the needs of the implementation', and the third one referred 'see above: web service complexities'.

[2]What they meant with this we did not fully understand

8.5 Evaluation

In the previous sections, and in Chapter 7, we collected evidence that will help us to evaluate to what extent our design satisfies our design requirements of Chapter 1, as given in Tables 1.1 (functional requirements), 1.2 (non-functional requirements w.r.t. development) and 1.3 (non-functional requirements w.r.t. use). We now do this evaluation, where we follow the grouping of requirements in these tables.

8.5.1 Functional requirements

Ad 1: the tool should be based on ioco theory Clearly, the tool is based on the **ioco** theory. This shows for example in the DerivationEngine interface signature (Table 3.8). Moreover, extensions are incorporated in the design: support for **uioco** (Section 4.2.6), for divergence (Section 4.2.7), and for symbolic models (Chapter 6).

Ad 2: the tool should work on models that have an LTS semantics Clearly, the tool works on models that have an LTS semantics, as shown in the signature of the Explorer interface (Section 4.2.2), and in Section 4.2 where we discuss how we provide access to models that have an LTS semantics.

Ad 3: the tool design should be suitable for both on-line and off-line testing In Section 3.4 and Section 3.5 we discuss how we support random resp. guided on-line testing, and in Section 3.6 we discuss how we can support off-line testing. In case studies we typically used on-line testing (Section 8.1); in the Conference Protocol Entity case study we also used TorX for off-line test execution of test cases which were derived using TGV, as mentioned in Section B.1. The master assignment reported in [Men08] used TorX to derive off-line test cases.

Ad 4: the tool should support on-line testing In Section 3.4 we show the Manager algorithm for on-line testing. In Section 4.2 we show how the Primer provides support for on-line testing. In case studies we typically used on-line testing, as discussed in Section 8.1.

Ad 5: the tool design should be independent from particular modelling languages In Section 3.4 we delegate model access to the DerivationEngine component. In Section 4.2 we decompose the DerivationEngine component in a modelling-language specific component, and a modelling-language-independent, generic, component. In Section 4.2.4 we list some of the modelling languages for which we have support. The case studies have used many different modelling languages. Researchers have been able to add support for an additional modelling language (Section 8.2).

Ad 6: the tool should support very large and infinite state space models Using on-the-fly access to the model, as provided by the Explorer interface (Section 4.2), allows the tool to deal with models that have an infinite state space (as long as they are finitely branching). We used this in the Conference Protocol Entity case study, see Section 8.1.1, and in the Software Bus case study, see Section 7.4. We also used other techniques to deal with large models: in the Easylink case study (Section 8.1.2) we used two techniques, an ad-hoc one, and one that uses symbolic modelling features of our support for Promela, to deal with an initial state with a huge number of outgoing output transition.

Ad 7: for on-line testing, the tool should support random mode and guided mode In Section 3.4 we discuss our support for random on-line testing, and in Section 3.5 our support for guided on-line testing. In all case studies we used on-line random testing mode; in a number of them we also used guided mode (see Section 8.1).

Ad 8: the tool design should make no assumptions about the SUT, except that it is a reactive system In Section 3.12 we discuss our Adapter interface, which is rather general and does not make any assumptions about the SUT. In Section 8.1 we describe some of the systems that we have tested using tools based on the design described in this thesis. This includes systems accessed over TCP (Sections 7.5.1, 7.5.2) and UDP (Section 8.1.1), on their standard input- and output (Section 8.1.1), via a serial line to a "magic black box" (Section 8.1.2), via a USB connection to a Lego NXT (Section 8.3.2: ToLERo BSc assignment), via a "human operator" (Section 8.1.2), and via test environments that hid much of the intricacies (Sections 8.1.4, 8.1.5).

Ad 13: it should be easy to create a simple model (like an automaton) for use with the tool In Section 4.2.4 we describe our support for modelling languages that make it easy to create simple models. In the responses to the questionnaire users (students doing the lab class of the Testing Techniques course at UT) indicated that indeed the modelling was easy (Section 8.4.2).

Ad 14: the tool should provide insight in the theory and algorithms that it implements, e.g. by visualisation In Section 4.2.5 we discuss our support for visualisation. Responses to our questionnaire show that the students appreciate the visualisation (Section 8.4.3). Visualisation also has contributed to demonstrations given to explain model-based testing to members of industry (Section 8.2).

Ad 15: it should be possible to use a simulated model as system under test In the introduction to Chapter 5 we mention that we support this functionality, even though we do not discuss details. We used this functionality to compare Conference Protocol Entity models with each other (Section 8.1.1), and to check execution logs of the storm surge barrier control system (Section 8.1.4). In Section 8.3 we mention how we use this in education.

Ad 18: the design should allow use of modelling languages suitable for non-experts In our opinion, the graphical modelling supported by yEd (GraphML) is suitable for non-experts, even though it is only usable to model small systems. In demonstrations that aimed to explain to non-experts the concept of model-based testing (Section 8.2), (at least some of) the models that were used were in GraphML.

In addition, if adding support for another modelling language would be felt as beneficial, the torx-explorer interface allows this (Section 4.2.4), and it has been done (Section 8.2).

Ad 19: the design should allow use of modelling languages with suitable expressive power In Section 4.2.4 we describe our support for modelling languages with suitable expressive power. In the discussion of the case studies, we also discuss the modelling languages that we used. Experience from the case studies, and responses to the questionnaire indicate that, in general, the modelling languages that we support provide sufficient expressivity. That said, two things are felt missing.

1. symbolic treatment of both data and time together (and at the STS language support implementation level, better diagnostics and error messages).
2. high-level language constructs which allow creating a single model that contains all functionality, and allow enabling, disabling and parameterisation of such model, to obtain specific model instances from it, as needed for separate test runs. To give an example: in the Highway Tolling System case study (Section 8.1.3) macro processor m4 [KR77] was used with modelling languages LOTOS and Promela, to construct a single model in which e.g. bad weather behaviour could be disabled, and the number of concurrent connections could be chosen, for each individual test run.

Ad 20: it must be possible to validate the models, either in the tool, or using external tools For modelling languages like mCRL2, LOTOS, and Promela, validation support is present in their respective tool environments. In the Highway Tolling System case study (Section 8.1.3) a serious error in the design was found when the model (based on that design) was validated, In Chapter 7 we describe how we validated the mCRL2 model using simulation and model-checking. In the Verification Engineering course at University of Twente, students first create and validate a model, then construct the system, which they subsequently test (Section 8.3.1).

However, such validation support is not available for, for example, the automata in GraphML that we draw with yEd. For those latter models, JTorX contains limited validation support in the form of the integrated interactive simulator (Section A.2). Moreover, the utility that exports a (J)TorX log file in Aldebaran format (Section A.2) can export any non-symbolic finite model to Aldebaran format—in this way, validation support of CADP and LTSmin can be used to validate any non-symbolic finite model. Finally, the visualisations may also help to validate models. In one case, an error in a test purpose, which was given in the Jararaca regular-expression-style modelling language,

was found thanks to the visualisation of the automaton that Jararaca uses as representation of the regular expressions.

Ad 21: the tool should produce/keep test execution data for analysis Both TorX and JTorX produce log files that contain, for each executed test step, the model label that represents it, a timestamp, and, when provided by the Adapter, a representation of the concrete interaction with the SUT.

Ad 22: the tool should be correct We have to evaluate correctness at two distinct levels.

(i) at the level of the design, and

(ii) at the level of the tool implementations that implement the design.

Ad i: In the previous chapters we mentioned a number of proof obligations (i.e. on pages 118, 144, 150, and 194), but we have not given the proofs.

Ad ii: Correctness of the TorX and JTorX implementations has been tested manually. In addition, there is anecdotal evidence that the tools find errors (case studies, student lab class exercises). Finally, in the first test runs of each case study typically lots of test failures occur. Such failures are carefully analysed, and the error is typically caused by mistakes in model, Adapter, or even in the SUT.

8.5.2 Non-functional requirements w.r.t. Development

Ad 9: it should be easy to accommodate theoretical progress We illustrate to what extent our design fulfils this requirement by describing two changes: (i) a change in the **ioco** definition, and (ii) the addition of support for divergence (τ-cycles).

Ad i: Our initial design was based on the **ioco** definition in [Tre96]. In this definition of **ioco**, test cases are not input enabled, i.e. when the test case prescribes that a stimulus is to be applied, the tester is not willing to accept an observation. In our initial design, in automatic mode, in tester states in which both stimuli and observations were enabled, a random choice was made between applying a stimulus and obtaining and checking an observation, where each of these choices has a probability of a half. In [Gog03] Goga proposed a refinement to this random selection strategy. In this refinement he replaced the probability of a half by a probability that is chosen to better fit the (behaviour in the) model.

We refined our initial design, by making the choice between applying a stimulus and obtaining an observation based on more complete information. In the refined design we also check whether the Adapter has a pending observation, in addition to checking that stimuli and observations are enabled in the model, as we already did in our initial design. In the refined design the Adapter interface contained a obsPending() function that allowed the Manager to request whether the Adapter has a pending observation. (Obviously, this design contained a race: an observation might arrive at the Adapter just after it has informed the Manager that it has no pending observations. However, there was not much we

could do about that.) In this refined design, the main algorithm in the Manager was changed to use the obsPending() function.

Our current design is based on the **ioco** definition in [Tre08]. In this definition of **ioco**, test cases *are* input enabled, i.e. when a stimulus is to be applied, the tester is also willing to accept an observation. For this design, two changes were made to the Adapter interface. Firstly, the obsPending() function was removed from the interface. Secondly, the applyStimulus() function, which so far only returned a boolean value to indicate whether application of the stimulus had succeeded, was changed to return an action label. This action label could be either of two things: Either it was the stimulus that was applied, or it was an observation that was already pending (and that thus had prevented application of the stimulus). The Manager algorithm was then adapted to no longer use function obsPending(), but just derive the next test step based on information from the model, execute this test step (i.e. interact with the Adapter, which might return a pending observation), and then deal with the Adapter result.

Ad ii: In Section 4.2.7 we show the impact of dealing with τ-cycles. See also below, ad 10.

Ad 10: it should be easy to incorporate new conformance relations We illustrate to what extent we satisfied this requirement by discussing the impact of the addition of support for the **uioco** conformance relation. In the JTorX implementation of our design, we initially only implemented test derivation based on **ioco**; later we added support for **uioco**. The changes necessary to support **uioco** are described in Section 4.2.6.

Typically, changes to the conformance relation only have impact on the Primer component, and in the Primer component only on the expand() function that, for a given tester state, computes the set of possible stimuli and expected responses (together with the tester states reached by each of those)[3]. This was the case for the addition of support for **uioco**, as well as for the check for τ-cycles (support for divergence).

Ad 11: it should be easy to incorporate new test selection strategies In the discussion of the Highway Tolling System case study (Section 8.1.3) we show how the extension of the DerivationEngine interface with interface functions hasInputs and getInput, allows us to easily extend the architecture with new selection components, like the Selector component discussed there.

In addition, we have used the guidance functionality (Section 4.3) to use e.g. the traces produced by the iocoChecker tool to guide test runs. The fact that the guidance information is accessed on-the-fly during a test run means that we can also replace the guidance Explorer by a new tool component that implements a new selection strategy, e.g. one that is based on model coverage.

[3] There are a few changes in other places, to allow the user to choose between the supported conformance relations.

8.5.3 Non-functional requirements w.r.t. Use

Ad 12: it should be easy to deploy the tool (install and use) Our initial implementation, TorX (Section A.1), was relatively hard to install and use. TorX was hard to install, because it had to be installed from source, and it had quite a number of dependencies. Most of these dependencies were standard available on a Unix system, but installation on Windows necessitated installing of a Unix compatibility layer, as e.g. Cygwin. TorX was hard to use, because in TorX each component in the design was a separate program. Even though TorX did have a graphical user interface, configuration was done via textual configuration files, where each individual program had its own configuration file. The Unix shell could be used to combine these individual programs for a particular purpose. This made TorX extremely flexible for an expert user, but not so easy to use by the casual user, because configuration required detailed knowledge about the tool.

The second implementation, JTorX (Section A.2), was designed with ease of installation and use in mind. To install JTorX, it suffices to unpack an archive (e.g. a zip-file). Configuration of the tool is done via a graphical user interface. JTorX has built-in support for a few modelling languages, and built-in Adapter support for toy implementations. JTorX can also be used with a command line interface, which has proven to be helpful when executing large numbers of experiments. Ease of use and installation of JTorX has been confirmed by the respondents to our questionnaire, see Section 8.4.1.

Ad 16: it should be simple to connect the tool to toy implementations The built-in Adapter (mentioned in the introduction to Chapter 5) for programs that communicate using model labels, either over standard input and output, or over TCP, makes it easy to connect a toy implementation. This was confirmed by the responses to our questionnaire, see Section 8.4.4.

Communication of model labels over TCP has also been used by JTorX users to make the connection to their own Adapter, for example in the Software Bus case study (Chapter 7).

Ad 17: it should be possible to use the tool without being an expert in the theory that the tool implements To a certain extent, we can argue that the students that use the tool in courses, assignments and case studies are not experts in the theory that the tool implements. Their success, especially with internships and graduation assignments, shows that the tool is usable for them.

On the other hand, the Testing Techniques students do study the theory that underlies the tool implementation, before the lab class in which they use it. The Verification Engineering students, though, have not studied the underlying theory; they only get a demonstration before they have to use the tool.

Finally, we also see that the tool "works" in demonstrations, where the tool succeeds to convey insight in the **ioco**-based approach to model-based testing.

That said, there is one point where the tool does not succeed to hide the underlying theory: it offers the choice between the use of Straces (for testing

with **ioco**) and Utraces (for testing with **uioco**). To understand this choice, a relatively deep understanding of the underlying theory is necessary. Because **uioco** must be used when testing the UT Testing Techniques lab class exercise, in the lecture prior to the lab class special attention is given to **uioco**. In this regard, JTorX does not fully fulfil the requirement, but we do not see how this can be avoided, except by not offering this choice—but then it would no longer be possible to use JTorX to show the difference between testing with **ioco** and testing with **uioco**, which would be a pity. A possible work-around could be the introduction of a "user mode" choice, where "expert mode" would offer the choice between Straces and Utraces, and the default "casual user mode" would just use Utraces without offering the choice.

Ad 23: the tool should have sufficient performance to be usable During our case studies, in general we found that the tool has sufficient performance to be usable (Chapter 7, Section 8.1).

However, in the early case studies with TorX (Easylink, Highway tolling system), where we tried to use multiple modelling languages, we found that performance with LOTOS models was less than with Promela, to the point where only Promela was usable (see Section 8.1.2 resp. 8.1.3). In the Storm surge barrier controller case we initially struggled with performance (Section 8.1.4).

In the later case studies, with JTorX, performance appeared less of an issue. Of course, by then also the performance of the computer hardware had improved.

For the journal version of the Neopost case study paper, presented in Chapter 7, we obtained performance measurements with JTorX. In this case study we used the mCRL2 formalism for modelling, and accessed these models using the lps2torx tool from the mCRL2 tool kit. We observed that the testing speed is influenced by three factors: (i) the quiescence timeout value, (ii) the model, and (iii) the means to access the model. We discuss these factors below.

Ad i: quiescence timeout Clearly, when the tester chooses to observe while no observation is available, the tester has to wait until the quiescence timer expires. In the Neopost case, the quiescence timeout value was set to 100 ms. Even though this is not an extremely high value by itself, test steps in which it was not necessary to wait for quiescence happened a lot faster, as visible in Figures 7.14b, 7.15a, and 7.15b.

In the "Oosterschelde storm surge barrier controller" case (Section B.4) we used an ad-hoc approach, because setting the quiescence timeout value to the longest allowed system response time (85 minutes) would result in very long testing times.

Ad ii: model In the Neopost case we observed a clear correspondence between the amount of non-determinism in the model, the number of model states that belonged to each tester state, and the time that it took to test with the model. The higher the amount of non-determinism, the higher the number of model states per tester state, and the longer it took to do the testing.

Ad iii: means to access the model In the Neopost case we accessed the mCRL2 model using the lps2torx tool from the mCRL2 tool kit. We found that for a few test steps, the time that it took to compute the test step had a direct correlation with the number of steps that had already been executed. We

firmly suspected that the long computation time for these steps is caused by the state tables (sets, maps) that reallocate themselves to have more space, to hold the ever-growing number of states. Just to compare, we also used the lps2torx tool from the LTSmin tool kit. With this tool, the number of test steps that take more than twice the quiescent timeout time is much smaller than with the mCRL2 toolkit version.

Both lps2torx tool implementations use rewriting for the computations on the data values in the model. We noticed that the rewriter implementation (generic interpreter, or just-in-time compiled) makes a difference, where, obviously, this difference is more pronounced when there is more rewriting to do, i.e. the difference is bigger for the models that have more non-determinism.

Ad 24: it should be easy to connect the tool to the system under test According to the responses to the questionnaire, the effort, necessary to connect JTorX to a SUT, varies quite a bit: from easy to hard or very hard (Section 8.4.5). This is what the researchers respond, and this is confirmed by our own experience doing case studies. The researchers remark that this is mostly caused by intricacies in the SUT.

In three of the case studies (Highway tolling system, storm surge barrier controller, Myrianed protocol stack) we were able to use the testing environment developed by the developers of the SUT. In those cases it sufficed to develop an Adapter to connect to those testing environments—a task that was relatively trivial, especially compared to the efforts necessary to construct the respective testing environments themselves.

In our experience, connecting an Adapter to JTorX is rather trivial; more support for constructing the Adapter, e.g. for dealing with SUT intricacies, would be beneficial. The internship report [Mei12] contains suggestions for this.

8.5.4 Summary

In Table 8.3 we show to what extent, in our opinion, each of the design requirements is fulfilled.

For Requirement 22 we indicated '-/□' because of the omission of proofs w.r.t. the design, and the manual testing of the implementations (note that this does *not* mean that we have no confidence in the implementations!).

For Requirement 17 we indicated '+/□' because of the discussion about the Straces–Utraces choice.

For Requirement 24 we indicated '□' because of the very limited Adapter implementation support.

Nr.	*requirement*	*fulfilment*
1	the tool should be based on **ioco** theory	+
2	the tool should work on models that have an LTS semantics	+
3	the tool design should be suitable for both on-line and off-line testing	+
4	the tool should support on-line testing	+
5	the tool design should be independent from particular modelling languages	+
6	the tool should support very large and infinite state space models	+
7	for on-line testing, the tool should support random mode and guided mode	+
8	the tool design should make no assumptions about the SUT, except that it is a reactive system	+
13	it should be easy to create a simple model (like an automaton) for use with the tool	+
14	the tool should provide insight in the theory and algorithms that it implements, e.g. by visualisation	+
15	it should be possible to use a simulated model as system under test	+
18	the design should allow use of modelling languages suitable for non-experts	+
19	the design should allow use of modelling languages with suitable expressive power	+
20	it must be possible to validate the models, either in the tool, or using external tools	+
21	the tool should produce/keep test execution data for analysis	+
22	the tool should be correct	-/□
9	it should be easy to accommodate theoretical progress	+
10	it should be easy to incorporate new conformance relations	+
11	it should be easy to incorporate new test selection strategies	+
12	it should be easy to deploy the tool (install and use)	+
16	it should be simple to connect the tool to toy implementations	+
17	it should be possible to use the tool without being an expert in the theory that the tool implements	+/□
23	the tool should have sufficient performance to be usable	+
24	it should be easy to connect the tool to the system under test	□

Table 8.3: Fulfilment of design requirements. Good: +, ok: □, bad: -.

Chapter 9

Conclusion

In this chapter we present conclusions, discuss related work, mention a few possible extensions, and discuss availability of the tool implementations TorX and JTorX.

9.1 Conclusions

We started this thesis with a high-level summary of the goal of our work: "To design a flexible tool for state-of-the-art model-based derivation and automatic application of black-box tests for reactive systems, usable both for education and outside an academic context.", which we then broke down in a number of functional and non-functional requirements. From then on, we no longer looked at the high-level goal, but only referred to the requirements. Also the evaluation of our design, in Section 8.5, we did in terms of these requirements.

Now, we return to the high-level goal, and use the evaluation of Section 8.5 to reflect on the degree to which we achieved it. In the discussion below, we follow the structure of Section 1.2.2. After reflecting briefly on the functional requirements, we discuss the three high-level non-functional requirements: (a) the tool should be flexible, (b) it should be usable for education, and (c) it should be usable outside an academic environment. In Section 1.2.2 we extracted additional functional and non-functional requirements from these three high-level non-functional requirements.

Functional requirements The initial functional requirements, i.e. Requirements 1–8, have all been taken care of in the design (see Section 8.5).

Non-functional requirement a: Flexibility In Section 1.2.2 we translated "flexibility" as "evolvability": we wish our tool to be evolvable, which we made more specific in Requirements 9–11. Note that, although we speak of evolvability of the design, it is typically the implementation that is evolved.

In Section 8.5 we gave several examples of evolving of the tool by the tool author (e.g. enhancement of **ioco** support from the definition of [Tre96] to the

one of [Tre08], addition of support for **uioco**, support for divergence). In addition we mentioned enhancement of the tool by others: development of the SPEX Explorer for Promela by a MSc student, development of the STATECRUNCHER Explorer by people in industry (see Section 8.2), and extension of the tool with support for the "lazy-on-the-fly model-based testing" approach [Far14, Kut14] (see Section 8.2). These extensions all were successful, to our knowledge. On the other hand, one response to the questionnaire indicates that construction of another Explorer was "very difficult" due to "no good documentation" (see Section C.2)—unfortunately we do not have further information.

So, regarding evolvability we have mixed success: evolvability by others may need more attention, in particular w.r.t. documentation.

Non-functional requirement b: Usability for education In Section 1.2.2 we discussed two scenarios related to this requirement: firstly, use of the tool in courses, to allow the students to experience the concept of model-based testing, and secondly, use of the tool to explain, on an intuitive level, the basic principles of model-based testing to almost any person (but in particular: testers and managers). This we made more specific as Requirements 12–16.

- Use of the tool in courses, to allow the students to experience the concept of model-based testing, appear to be successful. This is our own experience, which is confirmed by the use of the JTorX tool for teaching by others.
- Use of the tool to explain the basic principles of model-based testing also appears to be successful. We base this on our own experience, and on the experience by others that are using JTorX exactly for this purpose.

Non-functional requirement c: Usability outside an academic context In Section 1.2.2 we discussed two scenarios related to this requirement: firstly, use by students for doing internships or external graduation projects, and secondly, use by people in industry.

We have seen successful use of the tool JTorX by students in internships and external graduation projects. We also have seen successful use of this tool by researchers in case studies. We have not seen use of this tool by people in industry, i.e. we lack information to evaluate that part of the requirement.

Conclusion Overall, we succeeded to fulfil our high-level requirements, with the following two critical notes: (i) we lack information about the usability of the tool by people in industry, and (ii) we should assess possible improvements to the documentation for developers.

9.2 Related Work

In this section we discuss related work. An overview of model-based testing tools can be found in for example [BFS05]; [UPL12] gives a taxonomy of model-based testing approaches.

TGV The tool TGV [JJ05] derives off-line tests, given a specification and a test-purpose. It implements the **ioco** implementation relation. It treats divergent states (states on τ-cycles) as being quiescent, like we do with the approach described in Section 4.1.3. TGV does not have a "random walk" strategy.

TGV is distributed as part of the CADP tool-set; the specification and the test-purpose can be given in any of the input formalisms supported by CADP that can be interpreted as an LTS. TGV does not have the two-dimensional verdicts like we described in Section 2.3, but it has verdicts **pass**, **inconclusive**, **fail**, **none** (the default verdict, like we use $\perp$), and **error**.

Where our test purposes only have goal states, the TGV test purposes have both "accept" states (comparable to our goal states), and "refuse" states, that can be used to explicitly cut out traces that might otherwise result from the auto-completion done by TGV: for each test purpose state that has no outgoing transition for each input action, a self-loop with the special action "*" is added. This "*" action represents any input action not explicitly given on an outgoing transition from that state. Our test purposes do no have such auto-completion; the closest that we have is the use of sets of actions in the JARARACA input language [Jar12] (JARARACA is discussed in Section 4.2.4), but these have to be added by hand—they are not automatically added.

Command-line options of TGV can be used to fine-tune what to generate—a single test case with, or without, loops, or a single graph that contains all test cases—how to generate it, and in what format (like e.g. TTCN or the CADP [CAD] binary coded graph format).

SpecExplorer SpecExplorer [VCG+08, spe] is a tool that extends Microsoft Visual Studio with functionality to visualize, validate and generate test cases from the models that can be created with it. It supports on-line and off-line testing, where for off-line testing users can provide their own strategy to traverse the transition system that is derived from the model. In SpecExplorer models can be written as "model program" coded in C#, and as "behavioural descriptions" coded in the Cord scripting language. The former describes the system's behaviour, and the latter allows the user to select relevant behaviour. The ability to compose models of either kind enables users to slice out test cases from large state machines. The behaviour description has one special feature: the ability to indicate so-called "accepting states": those states in which tests are allowed to terminate, such that the system under test is left in a "good" state.

STG In a way, the tool STG (Symbolic Test Generator) [CJRZ02, PJJ07, PP] "lifts" the approach of TGV to symbolic tests: STG derives symbolic off-line tests from a symbolic model and a symbolic test purpose—both an IOSTS (defined in [RdBJ00]); also the generated test case is an IOSTS. STG has its own modeling language for the specification and test purpose IOSTSes.

For execution, a generated IOSTS test case is translated to Java; the constraint solver of Lucky [JR04] is used to instantiate symbolic stimuli.

UPPAAL-TRON UPPAAL-TRON [LMNS05] is a tool for on-line model-based testing of real-time systems. It does its work given a model of the system behaviour, which is specified as a network of timed-automata, and an environment model, which fulfills a role that is very similar to the guidance in our approach. UPPAAL-TRON implements the **rtioco** relation. It has significant expressivity also on the data, but, whereas time is treated in a symbolic manner, data values are not (they are enumerated, as in LTS-based approaches).

TorXAkis TorXAkis is a tool for on-line model-based testing, developed by Tretmans to research symbolic testing [MPS+09]. TorXAkis implements the **sioco** implementation relation. It does random walks through the model. TorXAkis has also been used in a timed testing setting (next to JTorX and UPPAAL-TRON) in the The Myrianed Protocol case study described in Section B.5. In that case study, the testing environment allows the testing tools to control the clock of the IUT, i.e. testing happened in simulated time, not real-time. In the model for TorXAkis, time was modeled as yet another variable in the symbolic model.

Whereas the initial TorXAkis implementation relied on the functional programming language (Haskell) in which TorXAkis is implemented to enter a model into the tool, it now has its own input language, which allows models to be entered using process-algebraic notation.

BAiT The tool BaiT [Cal08] also does symbolic testing, but it uses a different approach than STG and TorXAkis. It uses TGV to generate test cases, but it does not invoke TGV directly on the system model, but on a version of the model from which all data has been abstracted away. From the model it also generates a constraint logic program (CLP), and during test execution this CLP is used to reintroduce the data, also for the expected outputs. However, during test execution a non-deterministic system may produce an output that is valid, but different from the one that was (re)introduced in the test case. To cope with such situation, BaIT contains a mechanism that uses the system model to adapt the test case, such that it can be executed further.

Axini TestManager Axini [axi] TestManager is a testing tool developed by model-based testing company Axini, for both on-line and off-line testing. It implements the **ioco** and **sioco** implementation relations: **ioco** for LTS-based models, and **sioco** for STS-based ones. It has strategies for automatic test-selection, including model-coverage-based ones (the latter ones at least for LTS-based models with a finite state space, and for STS-based models). Using these strategies, it typically derives and executes multiple (many!) test cases in a run, as specified by the user (or as necessary to achieve the requested model-coverage). Axini TestManager is a SAAS (sofware as a service) solution: users access it via a web-interface. Where the GUI of JTorX is limited to configuration and presentation of results of an individual test run[1], the Axini TestManager

[1] In JTorX, the only exception to this is the experimental support to measure model coverage, and to use the measured model coverage as guidance information in a next test run.

gives an overview of, and access to, all tests that are executed in a run.

Other tools and approaches TTCN-3 [WDT+11] is a standardized test scripting language. In Section 5.3 we discuss the TTCN-3 test system reference architecture for execution of (compiled) TTCN-3. An important feature of the TTCN-3 approach is the separation of concerns between the TTCN-3 tests (abstact test suites) on the one hand, and the execution environment on the other, such that the tests are fully portable, independent of any platform implementation. Tests in TTCN-3 can be written by hand; there are also test generation tools that generate tests in TTCN-3.

The UML Testing profile [UTP] provides extensions to UML to support the design, visualization, specification, analysis, construction, and documentation of the artifacts involved in testing. It is independent of implementation languages and technologies, and can be applied in a variety of domains of development.

The QuviQ QuickCheck tool [AHJW06] has an interesting feature: after finding a failure, it tries to "shrink" the trace to the failure, to find a (much) shorter trace to the same failure. Such functionality could be very beneficial with our approach too, especially because with the on-line testing approach we may easily do very long test runs, but analysis of the test results is not always easy.

Test selection techniques make up another interesting topic. In [Wei09] Weiglhofer describes various approaches to test selection, including derivation of coverage-based test purposes from LOTOS specifications. In a case study he applied these approaches on several systems, one of which was the chatbox of Section B.1, and he compared his results for the chatbox with the results that others published for the same system.

9.3 Possible Extensions

Here we discuss a few possible extensions to the tool, based on our own experience with the tool, on the experience gained in case studies, and on responses to the questionnaire.

Explorer extensions The current support for symbolic models is split over two separate Explorer instances: one for STSes, and one for a network of timed automata. Unfortunately, the Explorer for the network of timed automata only treats time symbolically. As expressed in the questionnaire by one of the researchers, a nice extension would be an Explorer that treats both data and time symbolically.

On a more practical level, the STS Explorer expects the model to be given in an XML file, which currently has to be written by hand. Therefore, as mentioned in the questionnaire by another researcher, another nice extension would be a domain-specific language for STS models.

Finally, JTorX currently has no built-in support for creation of models[2]. In

[2]Actually, the model animation tool contains experimental support for the creation of models, but this is part of the 'advanced' functionality, which is hidden by default.

the questionnaire, one of the researchers expressed the wish for an integrated model-creation front end.

Adapter extensions A library of Adapter instances, for example for GUI testing, would be helpful, as indicated by respondents to the questionnaire. The same holds for library support to ease Adapter construction, as mentioned in [Mei12].

Model coverage Currently, the JTorX tool contains basic functionality to measure model state- and transition coverage, and a heuristic that uses this coverage information to guide the test derivation to unvisited parts of the model (we did not discuss this functionality in this thesis).

This functionality could be extended, and the coverage-based guidance heuristic could be refined.

9.4 Availability

Both TorX and JTorX can freely be downloaded, and both are open-source. JTorX is being maintained; maintenance of TorX stopped several years ago.

TorX is distributed in source form, under the Apache License, Version 2.0.
It is available from [torc].
Maintenance stopped several years ago.

JTorX is distributed in binary packages for Linux, Mac OS X, and Windows, available at [JTob].

- The JTorX git repository [JToa] contains the source of the core components.
- The STSimulator Explorer component (included in the JTorX distribution) has its own source repository at [STS].
- For the Jararaca Explorer component (included in the JTorX distribution), the source is available as part of the TorX distribution.
- The lps2torx Explorer component is not distributed with JTorX. However, two instances are available, in resp. the mCRL2 and the LTSmin tool set. Both these tool sets are open-source; see their respective web sites [mCR] and [LTS].
- For other external Explorer components the availability varies; some are available as part of TorX.

JTorX is being maintained; the JTorX web site [JTob] has an issue tracker.

Appendix A

Implementations

In this chapter we discuss two implementations of our design: TorX and JTorX. TorX was our first implementation, in which we shaped our design. JTorX is a reimplementation of TorX, created with ease of deployment in mind.

In Section A.1 we discuss TorX, in Section A.2 we discuss JTorX, and in Section A.3 we present a table that lists the functionality of both TorX and JTorX.

A.1 TorX

The core of TorX was developed in the *Côte de Resyste* project [TB03b], which ran from 1998–2002. Effectively, the goal (of both TorX and the project) was to make testing theory, which had been developed in the years prior to the project, tangible. Work on the testing tool, that was to become TorX, had already started before the start of the Côte de Resyste project; the latest release of TorX is from October 2008. TorX was developed using a number of implementation languages, like C, Tcl, Tk, and Perl. It is distributed in source form. The total distributed source consists of 239k lines of code, documentation, and build scripts; however, this number includes several 10k's lines of code that have been generated. TorX has been available for public download at [torc] since December 2002, initially under a license that only allowed non-commercial use; in December 2006 the licence was changed to the Apache License, Version 2.0.

Development of the tool was driven by the case studies that were done during this project. TorX is extremely flexible, but the benefits of this were mostly restricted to its developers: for others it turned out to be relatively difficult to install and configure.

The flexibility of TorX comes from the following two reasons: (1) each of its architectural components is mapped onto a separate program; (2) these programs communicate over their standard input and output, using textual interface messages. This makes it easy to use the Unix shell to compose components in unanticipated ways, and to create and incorporate additional components.

The implementation as several independent components also made it easy to create utility programs that, on the one hand, implement the CADP [GLMS07]

Open/Caesar interface [Gar98], and on the other hand connect to an Explorer, Primer or Combinator, thus allowing the use of e.g. the CADP simulator and the CADP minimization tools on a model, suspension automaton, or the structure offered by the Combinator.

TorX can be used via a command line interface CLI), and via a graphical user interface (GUI). Configuration is done via per-component configuration files. The GUI contains custom, user-configurable, menu's for models, guidance models and implementations. These menu's make it easier to e.g. switch from one model to another one, or one implementation to another one, during demonstrations.

Because TorX is distributed in source form, to be able to install TorX, the target system needs support for all the languages that are used in the implementation of TorX. On a typical Unix system this support is already there; however, on Windows this is more of a problem; there, a user has to install a Unix compatibility layer, as offered by e.g. Cygwin [cyg], before starting the installation of TorX.

A.2 JTorX

JTorX was developed as a re-implementation of TorX, with focus on ease of use, with respect to both installation and configuration [Bel10]. Work on JTorX started approximately in 2008. The core of JTorX is developed in Java (Java was chosen with ease of deployment in mind). JTorX' visualization components—written in Tcl/Tk—are reused from TorX, packaged as a Tcl Starpack, and integrated in JTorX, together with the the TCLKIT program that is necessary to run them. JTorX is distributed in binary form, packaged for Windows, Linux, and Mac OS X; the source repository is also publicly accessible at [JTob]. JTorX has been available for public download at [JTob] since January, 2009, under the three-clause BSD license.

JTorX lacks some of the flexibility that is present in TorX: JTorX is essentially a single program, which has interfaces to connect external Explorer and Adapter programs.

In JTorX, configuration is done via a graphical user interface (GUI). Configurations can be saved, for later use with the GUI. JTorX can also be used as command line utility, without GUI, e.g. to run tests with previously saved configurations.

JTorX is distributed with Explorer support for ALDEBARAN, GRAPHML, GRAPHVIZ, JARARACA, STS, and for its own log files. Moreover, the mCRL2 and LTSmin toolkits contain support for JTorX, which provides support for MCRL2.

In addition to the model-based testing functionality, JTorX contains an interactive simulator, and a tool called iocoChecker.

- The interactive simulator can simulate a model, test purpose or model-to-be-used-as-SUT (then it is connected to the Explorer), the suspension automaton of a model (then it is connected to the Primer), or the combination of a model and a test purpose (then it is connected to the Combinator).

Progress through the model can be visualised, using the functionality discussed in Section 4.2.5.

- The iocoChecker tool was developed as a separate tool by Lars Frantzen [Fra], and has been integrated into JTorX. It can check whether an implementation model is **ioco**- or **uioco**-conforming to a specification model (see Chapter 2). It exhaustively checks, for each state of the specification, whether its set of enabled outputs includes the corresponding set for the corresponding state in the implementation. In case of non-conformance, it shows the trace to the model state, and the output sets for that state of specification and implementation. The trace can be used as guidance information in a test, and it can be used to guide the interactive simulator, when simulating the specification or implementation model.

JTorX is distributed with a utility that reads any non-symbolic finite model in one of the supported modelling languages (see Section 4.2.4), and writes it out in Aldebaran format. This allows analysis of these models, and of JTorX log files, using tools like CADP and LTSmin.

The distribution of JTorX is rather self-contained: for installation of JTorX, only a Java run-time environment is necessary. Thus, it is simple to install JTorX on Windows.

A.3 Synopsis

Table A.1 gives an overview of the functionality of TorX and JTorX.

	TorX	*JTorX*
Functionality		
on-line testing	✓	✓
off-line test derivation	✓	-
off-line test execution	✓	✓
iocoChecker	-	✓
uiocoChecker	-	✓
interactive simulator	-	✓
CLI	✓	✓
GUI	✓	✓
Configuration	per-component	via GUI
User-configurable GUI menu's	✓	-
Testing Relation etc.		
ioco	✓	✓
uioco	-	✓
τ-cycle detect	✓	✓
divergence	-	✓
Input Formats		
Aldebaran (.aut)	Explorer	built-in

GraphML (.graphml)	-	built-in
GraphViz (.gv)	-	built-in
FSP	Explorer	Explorer from TorX
Jararaca	Explorer	built-in
LOTOS	Open/Caesar Explorer	Explorer from TorX
LOTOS	SmileExp (external)	SmileExp (external)
μCRL	lps2torx (external)	lps2torx (external)
mCRL2	lps2torx (external)	lps2torx (external)
Promela	Trojka	-
Promela	SPEX	SPEX from TorX
Timed automata net	ta2torx (external)	ta2torx (external)
STS	STSimulator (external)	built-in
TorX log	-	built-in
Torx-explorer interface	✓	✓
Output Formats (to allow use of third-party tools on (J)TorX models etc.)		
Aldebaran (.aut)	-	in distr
Open/Caesar interface	in distr	from TorX
Accessible Interfaces		
Torx-explorer interface	✓	✓
Torx-adapter interface	✓	✓
Primer interface	✓	-
Combinator interface	✓	-
Adapter		
labels over stdin/stdout		built-in
labels over TCP		built-in
simulated model	via configuration	built-in
Torx-adapter interface	✓	✓
Visualization		
model	via configuration	built-in
guidance model	via configuration	built-in
implementation (sim. model)	via configuration	built-in
suspension automaton (SA)	via configuration	built-in
'test run' (visited part of SA)	via configuration	built-in
message seq. chart	via configuration	built-in

Table A.1: Overview of functionality of TorX and JTorX

Appendix B

Case Studies

B.1 Conference Protocol Entity

The *conference protocol* is a simple chat box protocol that was designed for a course on protocol implementation. We discussed the Adapter for the CPE in Section 5.1.2. We tested a number of conference protocol entity (CPE) implementations using models in LOTOS and Promela, as discussed in [BFdV+99]. We also used TorX to execute test cases that were derived from the LOTOS model with TGV [JJ05], discussed in [DRS+00]. In addition, we tested a LOTOS and mCRL2 model of the CPE against each other.

Test architecture For the test architecture and the Adapter, used to test the CPE implementations, see Section 5.1.2.

Test tools TorX and TGV; LOTOS support via CADP [Gar98], Promela support via Trojka [dT00].

Testing modes on-line with TorX, random (+manual); off-line with TGV

Models We used LOTOS, Promela and mCRL2 models. In each of the models, we modelled not only the protocol behaviour of the CPE, but also the possibility that concurrently sent messages by the CPE may overtake each other. In the LOTOS and mCRL2 models we used unbounded FIFO queues, and thus the LTSes of these models have infinite state spaces.

Results Both with the LOTOS and the Promela models TorX was able to detect **ioco**-incorrect CPE implementations. To our surprise, two CPE implementations in which we had introduced errors were still **ioco**-correct to the models, because the scenario, necessary to trigger their errors, did not occur, given our models (and corresponding test set-up).

When running the LOTOS model against the mCRL2 one, we found one error in the mCRL2 model, even though both models have an infinite state space.

Lessons learned

1. Number of test steps, needed to trigger errors very much depends on random number generator seed.
2. Analysis of long error traces is cumbersome; translation of these traces to format that allows study using existing tools would be helpful.

3. We should not overlook unwanted (bad-weather) behaviour, even when we focus on expected (good-weather) behaviour.
4. Producing the Adapter is a laborious task.

Impact on design and implementation Translation of test logs to the Aldebaran format has been added (lesson learned 2).

Design requirements covered We only mention specific requirements.

1. Requirement 3: execution of test cases derived off-line by TGV;
2. Requirement 5: formalism independent: LOTOS, Promela and mCRL2 models;
3. Requirement 6: tool supports infinite state space models, i.e. the LOTOS and mCRL2 ones;
4. Requirement 15: support for simulated model as SUT allowed testing of LOTOS and mCRL2 models against each other;
5. Requirement 20: the LOTOS model was validated using simulators in LITE [BLV95] and CADP, and the Promela model using Spin.

Design requirements triggered

1. Requirement 21: produce/keep test execution data: translation to tool-supported format was triggered here.

Publications The case study was reported in [BFdV+99, DRS+00].

B.2 EasyLink

The EasyLink protocol facilitates communication between a TV and one or more Audio/Video (AV) devices, like VCRs. We tested its preset download feature, which allows automatic downloading of predefined settings (e.g. channel number, frequency, etc.) from the TV to AV devices.

Test architecture We tested a TV that was connected to a single VCR via an intermediate device called MBB by means of scart cables. The MBB—a proprietary device, developed within Philips for testing purposes—is connected to, and controlled by, a networked computer running the test tool (TorX) via a bidirectional serial link. The MBB takes care of all timing constraints of the scart communication. The TV can be operated by a uni-directional remote control. The remote control is also controlled by the computer via a human interface—a human that presses buttons on the remote control, as instructed by the computer.

To best use the computational resources which were available for this case study—relatively slow machines—a distributed testing tool set-up was used: test derivation took place on one machine, and the Adapter that interacted with the MBB ran on another machine. This was facilitated by the TorX Adapter interface, that, for this case, was run over TCP.

Test tool TorX; Promela support via Trojka [dT00].

Testing modes on-line, random

Model In Promela, with an extension to allow parameters (without constraints) in labels. The label parameters were used to deal with the unknown initial state—its list of presets—of the TV. The TV that was used, has a preset list of 100 entries, each containing a name and a frequency. So, without the parameters in the labels, the enumeration of all possible responses at the start of the test run—all possible combinations of possible names and frequencies, for 100 presets—would have given a huge number of expect outputs. With the parameters in the labels, this number was manageable.

This approach of using labels with parameters was one of two approaches which we tried to tackle the "unknown initial TV state" problem—for the other approach we refer to [BFHd01].

Results Two (believed) errors were detected.

Lessons Learned

- A generic method was used to deal with SUT output messages that were deemed irrelevant, such that they did not have not appear in (and clutter) the model, but still did appear in the test log.
- The ability to distribute the testing tool over multiple machines was very useful;
- User guidance in the form of test purposes (i.e., specifications that specify the property that is to be tested) would greatly improve the usability of the tool in a practical setting.

Impact on design and implementation

- introduction and application of symbolic testing to decrease the state space, in particular to reduce the number of transitions;
- the ability to distribute the tester over multiple machines;
- need for user guidance identified.

Design requirements covered

- Requirement 19: modelling language with expressive power, suitable to express large number of transitions;
- Requirement 23: in this case, the tool had sufficient performance to be usable, even though distribution of the tool functionality was necessary.

Publications The case study was reported in [BFHd01].

B.3 Highway Tolling System Payment Box

In the *highway tolling system* a money transaction takes place, to pay toll, when a car drives through a tolling gate.

We tested whether the *payment box* (PB), the tolling gate component that participates in the money transaction protocol, correctly implements the money transaction protocol. We used multiple concurrent transactions, and included the possibility that protocol messages getting lost.

Test architecture We tested the actual PB hardware and software. However, because the transactions are encrypted, and we did not have access to the keys, we could not create an Adapter that directly interfaces with the PB. We were able to use a testing framework that had already been developed for traditional testing of the PB. It interacts with the PB over UDP. We added a "remote control" interface—to be accessed over TCP/IP—to this testing framework, to be able to apply stimuli and obtain responses—where the testing framework took care of encryption and decryption. An Adapter provides the connection between TorX and the remote control interface.

Challenges When the PB waits for a next message in a transaction, it only waits for t_0 seconds (100ms); if no message has arrived by then, the PB aborts the transaction with a TimeOut message. This makes it hard to decide on a suitable quiescence timeout value t_δ: a value $>= t_0$ will cause transaction time-outs; a value $< t_0$ may result in unjustified test failures. Therefore, we choose $t_\delta > t_0$, and let the Adapter, when asked for an observation, return special action Tick when the PB is waiting for input, and no other output is available. This use of Tick is included in the model.

When testing with multiple concurrent transactions, it turned out that the change to successfully complete a transaction is very low, because apply inputs that belong to the same transaction in time t_0. Therefore, we let the Adapter do

To deal with a stringent timing requirement of the PB—two particular protocol messages have to arrive within a given short time interval, or else the PB times out—we let the Adapter do action refinement: The model contains a special label that represents the sequence of the two messages. When the Adapter is given that label as stimulus, it lets the testing framework send the two messages in quick succession.

Test tool TorX; Promela support via Trojka [dT00].

Testing modes on-line, random, random+weights. We used a new selection strategy, random+weights, that associates weights with labels, and uses the weights during the random selection, to affect the probability that particular labels are chosen. We used it here to test with erroneous input, such that the erroneous inputs are tested less often than the correct ones.

Model In LOTOS and Promela, using macro processor m4, to easily instantiate the number of concurrent transactions, and to include/exclude partial behaviour descriptions.

Results One unexpected observation of quiescence was made; otherwise, no errors were detected.

Lessons Learned Action refinement may help to meet timing requirements of the implementation.

Impact on design and implementation For this case study, we added the Selector tool component to the architecture, see Section 8.1.3.

Design requirements coverage

- Requirement 11: for this case study, we extended TorX with a component that implements the random+weights selection strategy[a];
- Requirement 19: Expressive power of LOTOS and Promela, though sufficient to describe the system behaviour, does not allow parameterising the models for e.g. the number of concurrent transactions, or to include or exclude partial behaviours;
- Requirement 23: for this case study, we had to use action refinement in the Adapter to meet IUT's timing requirements—with faster test derivation that may not have been necessary.

Publications The case study was reported in [dBF02].

[a] This is not present in JTorX.

B.4 Oosterschelde Storm Surge Barrier Emergency Closing System

The Oosterschelde storm surge barrier emergency closing system (ECS) controls the main operation of closing and opening the barrier. Typically, in response to just a few inputs, the ECS produces a series of outputs, where each of the outputs only arrives after a specified amount of time has passed. The shortest expected response time (delay) is two seconds, the longest expected response time is 85 minutes. There are no timing requirements on inputs.

The ECS was tested using model-based testing, where we tested not only functional behaviour, but also experimented with testing the timing requirements on the outputs of the ECS. After the ECS was put into production, an event log was obtained from it, and transformed to a trace of model actions, which was treated as a special kind of SUT, to test whether the trace was valid.

Test architecture Normally, the ECS runs on dedicated hardware; it interacts with its environment using signal lines that have either a high or a low signal. For traditional testing of the ECS, a testing framework had already been developed, which allows testing of the ECS on an ordinary PC—it simulates all connections that the ECS has with its environment. For model-based testing, an Adapter was developed to use this testing framework. The Adapter received every second the status of all signal lines from the framework, via shared memory—if the Adapter looks "too late" a value may have been overwritten. This Adapter reported, to the Manager, the signal line values of all signal lines, but only when at least one value had changed.

With each applied stimulus, and each reported response, the Adapter includes a timestamp in the label.

To deal with the large difference in response time, which made it very hard to choose a sensible quiescence timeout value, we effectively tested without checking for quiescence. The Adapter was made to return a special label Tick when an observation was requested, and none was present.

Test tool TorX; Promela support via Trojka [dT00], guidance via jararaca.

Testing modes on-line, random, guided; IOCO, practically without quiescence.

Model We modelled the ECS in Promela. We modelled both functional behaviour, and the time delays for the outputs. We modelled the functional behaviour in terms of individual signal line changes. We modelled time delays as integers. We used a special process in the model to combine the individual signal line changes and delays into (expected) output labels, where also the special label Tick was included.

Scenarios We used Jararaca test purposes to guide the testing, to avoid applying many inputs while waiting for an output that only comes after a long time.

Results No errors were detected. The event log obtained from the running system was found to be a valid trace of the model.

Lessons Learned Since this was our first experiment with timed testing, we tried several approaches to deal with time. The approach where we only gave **fail** verdicts for functional errors, and logged timing errors, without stopping test runs due to a timing error, was most successful. For the other approaches we refer to [Bel02].

Impact on design and implementation

- To use the label, returned by the Adapter after applying a stimulus, to compute the successor state—in this case study that label contained a timestamp.
- Visualisation/animation of test purposes was added.
- (indirectly) timed testing was studied theoretically, and a timed Explorer for TorX was developed.

Design requirements covered

- Requirement 7: both random and guided strategies were used;
- Requirement 15: the trace, obtained from the execution log, was used as SUT;
- Requirement 19: after some tweaking, we got sufficient performance to not loose observations;
- Requirement 23: performance sufficed to not miss responses in the Adapter.

Publications The case study was reported in [Bel02].

B.5 Myrianed Protocol Entity

The Myrianed gossip Medium Access Protocol *gMAC* is a protocol, developed by CHESS, that allows CHESS Myrianed Wireless Sensor Network (WSN) nodes to communicate with each other, such that application programs that run on the WSN nodes can exchange messages. Communication is via radio, and, to save power, the radio is switched off most of the time. Important aspects of the protocol are time and nondeterminism: both play a role in the mechanisms that allow WSN nodes to synchronise their clocks—and thus, the periods in which their radio receiver is switched on.

We tested the gMAC protocol stack on a normal PC, using simulated time. This was part of a case study in which also two other model-based testing tools were used to test the same SUT: TorXakis, and Uppaal-Tron. Here we focus on the experience with JTorX, for a more elaborate discussion of the case study see [TV11].

Test architecture Normally, the gMAC protocol stack runs on the special hardware of the WSN nodes. For the case study, a test framework was developed by CHESS that allows to run and test the gMAC protocol on a normal PC, using simulated time. Testers can control the clock that is implemented by the test environment, by issuing clock tick commands that make time progress.

For JTorX we developed an Adapter to interact with this test framework.

Challenges The main challenge of this case study was to make the model. We used an iterative approach to obtain system knowledge from an expert (and from system source code):

1. we made a model that reflected our understanding of the system,
2. we tested the system w.r.t. this model, which typically resulted in **fail** verdicts, after which
3. we discussed the result with the expert, which resulted in a better understanding of the system, after which
4. we go back to step 1, and update the model to reflect our increased understanding, etc.

Test tool JTorX, using the Explorer for Timed Automata.

Testing modes manual, automatic (random)

Model Timed automata

Results Improved understanding of the IUT, obtained by discussing test results with the expert.

Lessons Learned

- Symbolic treatment of data (in addition to the already present symbolic treatment of time) would be beneficial.
- Support for data functions (as present in Uppaal, but not supported in the Explorer) would be beneficial.

Impact on design and implementation

Design requirements coverage

- Requirement 5: the design should be modelling language independent: here we were able to use a timed automata model.
- Requirement 19: the expressivity of the timed automata language, supported by the Explorer sufficed for the case study; nevertheless, support of data functions would have been beneficial.

Publications The case study was reported in [TV11].

B.6 Rivercrossing Puzzle Program

The rivercrossing puzzle program is used in the lab class of a course name Testing Techniques at University of Twente. The program implements the well-known puzzle in which a farmer wants to cross a river with a goat, a cabbage and a wolf. For each move in the puzzle, the player can say what the farmer takes with him when he crosses the river. The puzzle program has invalid moves, unsafe moves, and valid moves. The model is underspecified and non-deterministic. It is underspecified, because in an unsafe or invalid state, only output is specified: the error message that the puzzle program should present. It is non-deterministic, because each unsafe state has a transition to the initial state, and a transition back to the last safe state. This combination of underspecification and non-determinism makes that testing with **ioco** may lead to undesired **fail** verdicts: for testing **uioco** must be used.

The students fill-in-the-blanks in a given Java implementation of the puzzle program, and create (**uioco**-)incorrect mutants of it. They also create mutant models, both **uioco**-correct and **uioco**-incorrect ones.

Test architecture The puzzle program provides two interfaces to each environment, both on standard input and output: firstly, using labels of the model, and secondly, using a very simple encoding onto bytes. For the first interface, one of the JTorX built-in Adapter programs can be used; for the second interface, the students have to fill-in-the-blanks in an Adapter that connects to JTorX using the torx-adapter interface.

Test tool JTorX

Testing modes on-line, manual, random, guided.

Model The model is made using yEd, in GraphML.

Scenarios The students design their own scenarios, e.g. to expose the errors they introduced.

Results Typically, most of the erroneous implementations are uncovered.

Lessons Learned It was rather surprising that such a simple exercise already made testing with **uioco** necessary.

Design requirements covered

- Requirement 7: both random and guided strategies were used;
- Requirement 15: in the lab class, the students also test models against each other;
- Requirement 14: most of the students use the visualisation to see what happens during testing;
- Requirement 18: the modelling formalism is easy to understand, and suitable for this case, because the model is small enough;
- Requirement 10: use of the **uioco** conformance relation was necessary;
- Requirement 23: performance sufficed to not miss responses in the Adapter.
- Requirement 12: the students install and use JTorX themselves.
- Requirement 16: the students can use the built-in Adapter for toy programs.

Publications None.

Appendix C

Questionnaire

To obtain feedback about TorX and JTorX, we asked users to fill out a questionnaire. We received 13 responses.

Below we present the questions and answers of the questionnaire. For this presentation

- we have edited the questions to make them shorter;
- we have combined and summarised responses;
- where appropriate, we have combined related questions;
- where possibly relevant, we indicated the category of respondent (student, researcher).

Note that for a number of multiple-choice questions, a single respondent could select multiple answers.

We obtained responses in two "rounds":

- in fall 2013 we presented the questionnaire to an initial small group of users, and
- in spring 2014, after the Testing Techniques (TT) course at University of Twente, we presented it to the TT students, and to a few more researchers.

Just before the start of TT course a new version of JTorX was released, for use by the TT students, in which some of the feedback, obtained from the initial respondents, had already been taken into account. This may explain why some of the responses to the questionnaire seem to contradict each other—different respondents based their responses on their experience with different versions of JTorX, at different moments in time.

Some responses may refer to issues, present in some older versions of JTorX, that have been dealt with, and are no longer present in the most recent version.

Did you ever install JTorX? Platforms?	Windows: 11, Mac OS X: 3, Linux: 2
Installing JTorX was ...	Very easy: 4, Easy:8, Ok: 1
How much time took installing JTorX?	30 sec – 30 min, typical 5-10 min
Did you have to install additional software?	X11: 1; two 32-bit libs: 1; yEd, GraphViz: 2; CADP: 1.
Did you ever install TorX? Platforms?	Linux: 1, Solaris: 1
Installing TorX was ...	Impossible: 1, Hard: 1
How much time took installing TorX?	several hours; (long ago) 1 hour, student needed 1 day
Did you have to install additional software?	a lot; some – both don't remember details
Remarks about installing (J)TorX?	Switching from TorX to JTorX made installation much more comfortable, and hence also teaching and supervision.
How did you use (J)TorX?	in a course: 11 (as lecturer: 1), internship / graduation assignment: 2, case study: 5, research: 2
In what role did you use (J)TorX?	student: 10, researcher: 3, teacher: 2, other: 1 (developer)
Did you use the JTorX GUI?	yes: 13, no: 0
Using the JTorX GUI was ...	ok: 8, easy: 5
Did you use the JTorX command line tool?	yes: 1, no: 12
Using the JTorX command line tool was ...	very easy: 1 (helpful, easy to extend, extremely important for large experiments)
Did you use visualisations?	yes: 10, no: 1, N/A: 1
Which ones did you use?	model: 9; test run: 6; sim model as IUT: 4; msc: 4; real impl: 1; other: 1 (lazy OTF MBT)
Were they helpful?	yes: 8, no: 1, N/A: 4
How?	get insight; see what goes on; find errors; location in specification; show what part of model is visited
Possible improvements?	More userfriendly it should be.
Which viz. did you miss?	Statistics/plots over multiple runs, but that should be part of a large test/case study management tool. Monitoring

Table C.1: Overview of the questionnaire questions and answers about installation, respondent background, GUI and CLI, and visualisation.

How did you make a model?	yEd: 11, text editor: 3, graph drawing tool: 2 (1: .dot), modelling env: 2 (1: mCRL2, cadp), other: 1: programmatically, and transforming model-code into XML
Which modelling languages?	GraphML: 10, mCRL2: 3, STS: 2, Aldebaran: 1, GraphViz: 1
Making the model was ...	hard: 2, ok: 4, easy: 5, very easy: 1 hard: not modelling, but finding/using additional tools (researcher) hard: limitations in error messages for STSs (researcher)
How much time took modelling?	few days (case study student), 5 min – several hours (researcher), a lot (researcher), day per model (researcher); 10 min – 1.5 hours (students)
Modelling language expressivity?	ok (students); no data (researcher); no real-time in STS (researcher); tool limitations necessitate modelling restraints (researcher)
What add to modelling language?	data (researcher); DSL for STS (researcher).
Which modelling language to add?	NuSMV, because it is a very powerful modelling language
Tried adding a modelling language?	yes: 1
What was your experience?	very difficult, no good documentation
Remarks about modelling?	researcher: Error messages when JTorX cannot read the model are often not very helpful
Have you used test purposes?	yes: 6
Modelled in?	GraphML: 5, Aldebaran: 1, mCRL2: 1, STS: 1, other: 1: extension to combine multiple test purposes
How?	yEd: 6, text editor: 1, other: 1: programmatically
Making them was ...	hard: 1, ok: 1, easy: 2, very easy: 2 hard: a single test purpose for a single feature was simple, but combining became quickly complex (researcher)
How much time to create?	about 2 days, a student of mine about 2 weeks (researcher); no time at all (previous models) - 10 min (students)

Table C.2: Overview of the questionnaire questions and answers about modelling and test purposes.

What IUT did you test?	simulated model: 11, toy implementation: 6, real implementation: 7
How did you connect JTorX?	simulation of given model – directly connected: 9 real program, communicating labels on stdin/stdout: 7 real program, communicating labels over TCP: 1 real program, started via given TorX adapter: 7
Connecting was ...	hard: 2, ok: 7, easy: 3, very easy: 1 hard: sometimes easy, sometimes difficult (researcher) hard: you really need to know which connections need to be made (student) very easy: there were little problems with the custom adapter (student)
How much time took it?	a day or so (student case study); Practically nothing – 2 hour (students) from 0 (simulated model) to couple of hours, depends very much on SUT (researcher) all in all about a person week (a student, a developer from industry and I) (researcher) the simulations were easy to do; communication via TCP was a bit more iffy (student)
Did you create an adapter?	yes: 8
How did you connect JTorX?	simulation of given model – directly connected: 2 real program, communicating labels on stdin/stdout: 3 (1: was the easiest way) real program, communicating labels over TCP: 2 real program, started via given TorX adapter: 3
Connecting JTorX was ...	very hard: 1, hard: 1, ok: 3, easy: 1, very easy: 2
How much time took it?	researcher: about an hour (a student of mine did it and told me) Practically nothing. 2 hours
Creating an adapter was ...	very hard: 1, hard: 1, ok: 3, easy: 1
How much time took it?	researcher: all in all about 3 days (most work was a facade to hide all the web service complexities from the model and core adapter) 2 hours
Biggest stumbling blocks?	researcher: all the small details you have to consider, mostly due to the SUT researcher: to adjust the adapter for the needs of the implementation researcher: see above: web service complexities student: I didn't fully understand how it works. student: No clear documentation of the responsibilities of the different methods (in Java).
How to ease adapter creation?	researcher: a few example adapters and a better documentation on how to connect an adapter researcher: Offer a utility api.
Remarks about connecting JTorX to IUT?	researcher: better documentation

Table C.3: Overview of the questionnaire questions and answers regarding connecting JTorX to IUT.

What are the top three things you dislike about JTorX?
Sometimes hard to understand how several features work.
Need own code (adapter) to make it work.
no data
no integrated front end
option boxes in GUI without on-line help for which I always forget their meaning
very little documentation
buggy, it sometimes simply crashes
storing configurations doesn't work
some bugs, but none too difficult to fix
the core engine and STS integration
that lazy OTF MBT hasn't become a part of JTorX yet ;)
Sometimes too verbose tool-tips
sometimes too high window (testing mode)
the simulation windows look a bit ugly on Linux (but are fully readable).
Too many things on the screen.
several options and boxes that can be ticked, but for which I have insufficient knowledge of FMT to figure out what they do on my own.
A slightly clunky GUI that performs decently but is otherwise not immediately intuitive.
The lack of an in-depth manual to go with the tool
Hardness
not so much userfriendly
time consuming
What are the top three things you like about JTorX?
Very general
Seems useful in practice
Runs using Java
nice tool for playing, teaching, learning the theory
fast answer to questions
simple design (GUI is not overloaded)
visualisation of the test-process (msc, model ..)
the modular design and extensibility
the user-friendly GUI
that STSs are supported
It really works
the simulations are very informative
it can be customised and used in many ways.
Simple tool
Easy installment
the underlying theory
java-based
visualisation
The ease with which simulations from yEd are imported and tested
The range of communication possibilities, including TCP communication and Std in-/out labels

Table C.4: Overview of the questionnaire questions and answers about likes and dislikes.

What are the top three things would like to see added/changed/improved in JTorX? NuSMV support Function that uses GUI buttons as JTorX input (from implementation) in order to automate GUI testing. no data no integrated front end option boxes in GUI without on-line help for which I always forget their meaning handling of real-time and data more expressive STSs; a DSL for modelling STSs; more powerful core engine (symbolic, better solver, better guidance (e.g. via lazy OTF MBT)); Sometimes too verbose tool-tips, sometimes too high window (testing mode), the simulation windows look a bit ugly on Linux (but are fully readable). The manual, mostly! Especially for people using it in the course of the bachelor/master, without the extensive FMT background that some have, would be very nice to have!
Is there anything else that you would like to share about JTorX or TorX? JTorX has helped in my research a lot. Thanks. I really enjoyed working with this tool. :)

Table C.5: Overview of the questionnaire questions and answers about suggested improvements and other remarks.

Publications from the Author

[1] M. Sytema, A. F. E. Belinfante, M. I. A. Stoelinga, and L. Marinelli. Experiences with formal engineering: model-based specification, implementation and testing of a software bus at neopost. *Science of computer programming*, 80(Part A):188–209, February 2014.

[2] F. Arnold, A. F. E. Belinfante, F. I. Van der Berg, D. Guck, and M. I. A. Stoelinga. Dftcalc: a tool for efficient fault tree analysis. In *Proceedings of the 32nd International Conference on Computer Safety, Reliability, and Security (SAFECOMP), Toulouse, France*, volume 8153 of *Lecture Notes in Computer Science*, pages 293–301, Berlin, September 2013. Springer Verlag.

[3] F. Arnold, A. F. E. Belinfante, F. I. Van der Berg, D. Guck, and M. I. A. Stoelinga. Dftcalc: a tool for efficient fault tree analysis (extended version). Technical Report TR-CTIT-13-13, Centre for Telematics and Information Technology, University of Twente, Enschede, June 2013.

[4] A. F. E. Belinfante and A. Rensink. Publishing your prototype tool on the web: Puptol, a framework. Technical Report TR-CTIT-13-15, Centre for Telematics and Information Technology, University of Twente, Enschede, June 2013.

[5] M. Sijtema, M. I. A. Stoelinga, A. F. E. Belinfante, and L. Marinelli. Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at neopost. In G. Salaün and B. Schätz, editors, *FMICS 2011*, volume 6959 of *LNCS*, pages 117–133. Springer, August 2011.

[6] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Paphos, Cyprus*, volume 6015 of *LNCS*, pages 266–270, Berlin, March 2010. Springer Verlag.

[7] A. F. E. Belinfante. Extensible synthetic file servers? or: Structuring the glue between tester and system under test. In S. J. Mullender and

G. Collyer, editors, *Proceedings of the Second International Workshop on Plan 9 (IWP9 2007), Bell Labs, Murray Hill, NJ, USA*, pages 47–54, Murray Hill, NJ, USA, December 2007. Bell Labs.

[8] H. C. Bohnenkamp and A. F. E. Belinfante. Timed model-based testing. In G. J. Tretmans, editor, *Tangram: Model-based integration and testing of complex high-tech systems*, pages 115–128. Embedded Systems Institute ESI, Eindhoven, the Netherlands, 2007.

[9] A. F. E. Belinfante. Experiments towards model-based testing using plan 9: Labelled transition file systems, stacking file systems, on-the-fly coverage measuring. In G. Guardiola, E. Soriano, and F. J. Ballesteros, editors, *Proceedings of the First International Workshop on Plan 9, Madrid, Spain*, pages 53–64, Madrid, December 2006. Universidad Rey Juan Carlos.

[10] A. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *LNCS*, pages 391–438. Springer, 2005.

[11] H. C. Bohnenkamp and A. F. E. Belinfante. Timed testing with torx. In J. S. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Formal Methods Europe (FME), Newcastle, UK*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188, Germany, 2005. Springer-Verlag.

[12] A. F. E. Belinfante. Timed testing with torx: The oosterschelde storm surge barrier. In M. Gijsen, editor, *Handout 8e Nederlandse Testdag, Rotterdam, the Netherlands*, Rotterdam, 2002. CMG.

[13] R. G. de Vries, A. F. E. Belinfante, and J. Feenstra. Automated testing in practice: The highway tolling system. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, volume 210 of *IFIP Conference Proceedings*, pages 219–234, Dordrecht, 2002. Kluwer Academic Publishers.

[14] René G. de Vries, Axel Belinfante, and Jan Feenstra. Automated testing in practice: The highway tolling system. In Ina Schieferdecker, Hartmut König, and Adam Wolisz, editors, *Testing of Communicating System XIV*, pages 219–234, Berlin, 2002. Kluwer academic publishers.

[15] A. F. E. Belinfante, J. Feenstra, A. W. Heerink, and R. G. de Vries. Specification based formal testing: The easylink case study. In J. P. Veen, editor, *2nd PROGRESS workshop on Embedded Systems, Utrecht, the Netherlands*, pages 73–82, Utrecht, October 2001. Technology Foundation STW.

[16] R. G. de Vries, G. J. Tretmans, A. F. E. Belinfante, J. Feenstra, L. M. G. Feijs, S. Mauw, N. Goga, A. W. Heerink, and A. de Heer. Côte de resyste in progress. In J. P. Veen, editor, *1st PROGRESS workshop on Embedded Systems, Utrecht*, pages 141–148, Utrecht, October 2000. Technology Foundation STW.

[17] L. Du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. F. E. 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, *Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), Ottawa, Canada*, volume 176 of *IFIP Conference Proceedings*, pages 221–228, Dordrecht, August 2000. Kluwer Academic Publishers.

[18] G. J. Tretmans and A. F. E. Belinfante. Automatic testing with formal methods - samenvatting van de eurostar'99 presentatie. *TestNet Nieuws - Nieuwsbrief van de vereniging TestNet*, 4(1):8–10, 2000.

[19] G. J. Tretmans and A. F. E. Belinfante. Automatic testing with formal methods. Technical Report TR-CTIT-99-17, Centre for Telematics and Information Technology University of Twente, Enschede, December 1999.

[20] 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, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer, 1999.

[21] G. J. Tretmans and A. F. E. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review, Barcelona, Spain*, Galway, Ireland, 1999. EuroStar Conferences.

[22] P. van Eijk, A. F. E. Belinfante, E. H. Eertink, and H. Alblas. The term processor generator kimwitu. In H. Brinksma, editor, *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands*, volume 1217 of *Lecture Notes in Computer Science*, pages 96–111, Berlin, April 1997. Springer Verlag.

[23] P. van Eijk, A. F. E. Belinfante, E. H. Eertink, and H. Alblas. The term processor generator kimwitu (full version). Technical Report 96-49, University of Twente, 1996.

References

[AHJW06] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

[AHK+12] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F.W. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In *Proceedings 18th International Symposium on Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 27–31. Springer-Verlag, 2012.

[AKT+13] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer. Improving active mealy machine learning for protocol conformance testing. *Machine Learning*, pages 1–36, 2013.

[ald] Aldebaran .aut file format. `http://www.inrialpes.fr/vasy/cadp/man/aldebaran.html#sect6`.

[axi] Axini website. `http://www.axini.com`.

[BAL+90] Ed Brinksma, Rudie Alderden, Rom Langerak, Jan Tretmans, and Jeroen van de Lagemaat. A formal approach to conformance testing. In J. de Meer, W. Effelsberg, and L. Mackert, editors, *Second International Workshop on Protocol Test Systems*, pages 349–363. IFIP TC 6, North-Holland, 1990.

[BB04] Laura Brandán Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing: 4th International Workshop, FATES 2004, Linz, Austria*, volume 3395 of *LNCS*, pages 64–78, Berlin, September 2004. Springer Verlag.

[BB05] H. C. Bohnenkamp and A. F. E. Belinfante. Timed testing with torx. In J. S. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Formal Methods Europe (FME), Newcastle, UK*, volume 3582 of *LNCS*, pages 173–188, Germany, 2005. Springer-Verlag.

[BB07] H. C. Bohnenkamp and A. F. E. Belinfante. Timed model-based testing. In G. J. Tretmans, editor, *Tangram: Model-based integration and testing of complex high-tech systems*, pages 115–128. Embedded Systems Institute ESI, Eindhoven, the Netherlands, 2007.

[Bel02] A. F. E. Belinfante. Timed testing with torx: The oosterschelde storm surge barrier. In M. Gijsen, editor, *Handout 8e Nederlandse Testdag, Rotterdam, the Netherlands*, Rotterdam, 2002. CMG.

[Bel10] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Paphos, Cyprus*, volume 6015 of *LNCS*, pages 266–270, Berlin, March 2010. Springer Verlag.

[BFdV+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, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer, 1999.

[BFHd01] A. F. E. Belinfante, J. Feenstra, A. W. Heerink, and R. G. de Vries. Specification based formal testing: The easylink case study. In J. P. Veen, editor, *2nd PROGRESS workshop on Embedded Systems, Utrecht, the Netherlands*, pages 73–82, Utrecht, October 2001. Technology Foundation STW.

[BFS05] A. F. E. Belinfante, L. Frantzen, and C. Schallhart. Tools for test case generation. In M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *LNCS*, pages 391–438. Springer Verlag, 2005.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*. CWI, Amsterdam, The Netherlands, 1985.

[BLV95] Tommaso Bolognesi, Jeroen van de Lagemaat, and Chris Vissers, editors. *LOTOSphere: Software Development with Lotos*. Kluwer Academic Publishers, 1995.

[BPS01] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[Buu95] Kees Buurman. Vluis. *Vlieger*, 2, 1995. `http://www.kiteplans.org/planos/vluis/vluis.html`.

[BvdPW09] S. C. C. Blom, J. C. van de Pol, and M. Weber. Bridging the gap between enumerative and symbolic model checkers. Technical Report TR-CTIT-09-30, Centre for Telematics and Information Technology, University of Twente, Enschede, 2009.

[CAD] CADP web site. `http://www.inrialpes.fr/vasy/cadp/`.

[CAD12] CADP evaluator4 manual webpage. `http://cadp.inria.fr/man/evaluator4.html`, December 2012.

[Cal08] J. R. Calamé. *Testing reactive systems with data: enumerative methods and constraint solving*. PhD thesis, University of Twente, Wageningen, September 2008.

[CJRZ02] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 470–475. Springer Berlin Heidelberg, 2002.

[cyg] Cygwin website. `http://www.cygwin.com`.

[dAJ12] Roberto Alves de Almeida Junior. Model-based testing with a b model of the emv standard. Bachelor's thesis, Radboud University Nijmegen, July 2012.

[dBF02] R. G. de Vries, A. F. E. Belinfante, and J. Feenstra. Automated testing in practice: The highway tolling system. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, volume 210 of *IFIP Conference Proceedings*, pages 219–234, Dordrecht, 2002. Kluwer Academic Publishers.

[DRS+00] L. Du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. F. E. 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, *Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), Ottawa, Canada*, volume 176 of *IFIP Conference Proceedings*, pages 221–228, Dordrecht, August 2000. Kluwer Academic Publishers.

[dT00] R. G. de Vries and G. J. Tretmans. On-the-fly conformance testing using spin. *International journal on software tools for technology transfer*, 2(4):382–393, 2000.

[dV01] R.G. de Vries. Towards formal test purposes. In J. Tretmans, editor, *Formal Approaches to Testing of Software 2001 (FATES'01)*, pages 61–76, Aalborg, Denmark, 2001. BRICS, University of Aarhus, Denmark. BRICS Notes Series (NS-01-4).

[Eer94] Henk Eertink. *Simulation Techniques for the Validation of LOTOS Specifications*. PhD thesis, University of Twente, Enschede, The Netherlands, 1994.

[Far14] David Faragó. *Model Checking and Model-Based Testing: Improving These Formal Methods by Lazy Techniques*. PhD thesis, Karlsruhe Institute of Technology, 2014. forthcoming.

[Feea] Jan Feenstra. Conference protocol description. `http://fmt.ewi.utwente.nl/tools/torx/confcasedescr.html`.

[Feeb] Jan Feenstra. Conference protocol implementations. `http://fmt.ewi.utwente.nl/tools/torx/confcaseimpls.html`.

[FGM+10] Alessio Ferrari, Daniele Grasso, Gianluca Magnani, Alessandro Fantechi, and Matteo Tempestini. The Metrô Rio ATP case study. In Kowalewski and Roveri [KR10], pages 1–16.

[Fok00] Wan Fokkink. *Introduction to Process Algebra*. Springer, 2000.

[Fra] Lars Frantzen. Tools – iocochecker. `http://www.cs.ru.nl/~lf/tools/iocochecker/`.

[FTW05] Lars Frantzen, Jan Tretmans, and Tim A.C. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, number 3395 in LNCS, pages 1–15. Springer, 2005.

[FTW06] Lars Frantzen, Jan Tretmans, and Tim A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in LNCS, pages 40–54. Springer, 2006.

[Gab00] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, 2000.

[Gar98] H. Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 68–84. Springer Berlin Heidelberg, 1998.

[GKM+08] Jan Friso Groote, Jeroen Keiren, Aad Mathijssen, Bas Ploeger, Frank Stappers, Carst Tankink, Yaroslav Usenko, Muck van Weerdenburg, Wieger Wesselink, Tim Willemse, and Jeroen van der Wulp. The mCRL2 toolset. In *Proc. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, pages 5/1–10, 2008.

[GLMS07] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In *CAV 2007*, pages 158–163, 2007.

[Gog03] N. Goga. Experimenting with the probabilistic torx algorithm. In *International Workshop on Software Engineering for High Assurance Systems (SEHAS '03)*, pages 13–20, Portland, Oregon, USA, 2003. Software Engineering Institute, Pittsburg, USA.

[Graa] GraphViz dot language. `http://www.graphviz.org/content/dot-language`.

[Grab] GraphViz web site. `http://graphviz.org`.

[Gra12] GraphML file format. `http://graphml.graphdrawing.org`, January 2012.

[GVZ01] Hubert Garavel, César Viho, and Massimo Zendri. System design of a CC-NUMA multiprocessor architecture using formal specification, model-checking, co-simulation, and test generation. *STTT*, 3(3):314–331, 2001.

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

[HKL+10] H. H. Hansen, J. Ketema, S. P. Luttik, M. R. Mousavi, and J. C. van de Pol. Towards model checking executable UML specifications in mCRL2. *Innovations in Systems and Software Engineering*, 6(1-2):83–90, March 2010.

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

[Hol03] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[Int81] Internet Engineering Task Force. *RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification*, September 1981.

[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, Geneva, 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*, volume ITU-T Recommendation Z.500 of *Committee Draft CD 13245-1*. ISO – ITU-T, Geneve, 1996.

[Jar12] Jararaca manual. `http://fmt.cs.utwente.nl/tools/torx/jararaca.1.html`, January 2012.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[JR04] E. Jahier and P. Raymond. The lucky language reference manual. Technical Report TR-2004-6, Verimag, 2004.

[JTOa] JTorX git repository. `http://fmt.ewi.utwente.nl/gitweb/?p=jtorx.git`.

[JTob] JTorX website. `http://fmt.ewi.utwente.nl/tools/jtorx/`.

[KB02] Nitin Koppalkar and Animesh Bhowmick. Integration of generic explorer with the TorX tool chain. Technical Note 2002/387, Philips Nat. Lab., 2002.

[Kle12] Robert Kleinpenning. Is javacardsign correct and secure? Bachelor's thesis, Radboud University Nijmegen, July 2012.

[Kop03] Nitin Koppalkar. Interfacing STATECRUNCHER with TorX for demonstrating the state-based testing technique taking mg-r components for a case study. Technical Note draft report, Philips Nat. Lab., 2003.

[KR77] Brian W. Kernighan and Dennis M. Ritchie. The m4 macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey, USA, 1977. `http://wolfram.schneider.org/bsd/7thEdManVol2/m4/m4.pdf`. Accessed on May 19, 2014.

[KR10] Stefan Kowalewski and Marco Roveri, editors. *FMICS 2010*, volume 6371 of *LNCS*. Springer, 2010.

[Kut14] Felix Kutzner. A case study for lazy on-the-fly model-based testing. Bachelor's thesis, Karlsruhe Institute of Technology, 2014.

[Lan90] Rom Langerak. A testing theory for lotos using deadlock detection. In *Proceedings of the IFIP WG 6.1 Ninth int. Symp. on Protocol Spec., Testing, and Verification*, pages 87–98. IFIP, 1990.

[Leu13] Joost Leuven. Testing of channel based service connectors. Bachelor's thesis, Universiteit Leiden, August 2013.

[Lev00] Nancy G. Leveson. Experiences in designing and using formal specification languages for embedded control software. In Lynch and Krogh [LK00], page 3.

[LK00] Nancy A. Lynch and Bruce H. Krogh, editors. *HSCC 2000*, volume 1790 of *LNCS*. Springer, 2000.

[LMN05] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing*, volume 3395 of *LNCS*, pages 79–94. Springer Berlin Heidelberg, 2005.

[LMNS05] Kim G. Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: An industrial case study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 299–306, New York, NY, USA, 2005. ACM.

[LTS] LTSmin web site. `http://fmt.ewi.utwente.nl/tools/ltsmin`.

[mCR] mCRL2 toolkit website. `http://www.mcrl2.org/`.

[Mei12] Jeroen Meijer. Model based system testing in practice: Report on an internship performed at panalytical. Master's thesis, University of Twente, November 2012.

[Men08] Jeroen Mengerink. SeCo - a tool for semantic test coverage. Master's thesis, University of Twente, Enschede, The Netherlands, 2008.

[MPS+09] Wojciech Mostowski, Erik Poll, Julien Schmaltz, Jan Tretmans, and Ronny Wichers Schreur. Model-based testing of electronic passports. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, volume 5825 of *LNCS*, pages 207–209. Springer Berlin Heidelberg, 2009.

[MT08] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *Proceedings of the 15th International Symposium on Formal Methods FM'08*, volume 5014 of *LNCS*, pages 148–164, 2008.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979.

[Neo09] Neopost Inc. website. `http://www.neopost.com/`, August 2009.

[PJJ07] F. Ployette, B. Jeannet, and T. Jéron. Stg: a symbolic test generation tool for reactive systems. TESTCOM/FATES07 (Tool Paper), June 2007.

[Pos80] J. Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[PP] F. Ployette and F.X. Ponscarme. The STG Tool Page. `http://www.irisa.fr/prive/ployette/stg-doc/stg-web.html`.

[RdBJ00] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *LNCS*, pages 338–357. Springer Berlin Heidelberg, 2000.

[Rev] The shape that started a revolution. `http://revkites.com`.

[Roo86] Paul E. Rook. Controlling software projects. *IEEE Software Engineering Journal*, 1(1):7–16, January 1986.

[SBSM14] M. Sytema, A. F. E. Belinfante, M. I. A. Stoelinga, and L. Marinelli. Experiences with formal engineering: model-based specification, implementation and testing of a software bus at neopost. *Science of computer programming*, 80(Part A):188–209, February 2014.

[Sch05] Helen Schonenberg. Timed modeling and verification of the do/dg component: A case study for testing a real time component with torx, using verified timed models. Internship report, University of Twente, August 2005.

[Sni10] Arjan Snippe. ToLERo: ToRX-tested LEGO robots. Bachelor's thesis, University of Twente, 2010.

[spe] SpexExplorer website. `http://specexplorer.com`.

[Spe02] Robert L. M. Spee. Automatic test generation and execution in practice: TorX, TestFrame and the Easy Mail Machine. Master's thesis, Technische Universiteit Eindhoven, November 2002.

[SSBM11] M. Sijtema, M. I. A. Stoelinga, A. F. E. Belinfante, and L. Marinelli. Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at neopost. In G. Salaün and B. Schätz, editors, *FMICS 2011*, volume 6959 of *LNCS*, pages 117–133. Springer, August 2011.

[Sse06] Richard Ssekibuule. Model-based testing of network security protocols in java card applications. Master's thesis, Radboud University Nijmegen, October 2006.

[ST08] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In Franck Cassez and Claude Jard, editors, *Formal Modeling and Analysis of Timed Systems*, volume 5215 of *LNCS*, pages 250–264. Springer Berlin Heidelberg, 2008.

[Sto12] Gerjan Stokkink. Quiescent transition systems. Master's thesis, University of Twente, Enschede, The Netherlands, 2012.

[STS] Stsimulator project at java.net. `https://java.net/projects/stsimulator`.

[STS13] W. G. J. Stokkink, M. Timmer, and M. I. A. Stoelinga. Divergent quiescent transition systems. In M. Veanes and L. Viganò, editors, *Proceedings of the 7th International Conference on Tests and Proofs (TAP 2013), Budapest, Hungary*, volume 7942 of *LNCS*, pages 214–231, Berlin, June 2013. Springer Verlag.

[TB03a] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[TB03b] J. Tretmans and H. Brinksma. Torx: Automated model based testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proc. 1st European Conf. on Model-Driven Software Engineering*, 2003.

[tH12] Thijs ten Hoeve. Model based testing of a plc based interlocking system. Master's thesis, Technische Universiteit Twente, November 2012.

[The12] The Go programming language website. `http://golang.org/`, January 2012.

[Tho04] Graham G. Thomason. *The Design and Construction of a State Machine System that Handles Nondeterminism.* PhD thesis, School of Electronics and Physical Sciences, 2004. `http://freespace.virgin.net/graham.thomason/Statecruncher/StCrMainThesis.pdf`. Accessed on May 23, 2014.

[tora] torx-adapter(5) - a program that implements an interface to the sut. `http://fmt.ewi.utwente.nl/tools/torx/torx-adaptor.5.html`.

[torb] torx-explorer(5) - interface to program to explore a labelled transition system. `http://fmt.ewi.utwente.nl/tools/torx/torx-explorer.5.html`.

[torc] TorX website. `http://fmt.ewi.utwente.nl/tools/torx/`.

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

[Tre96] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996. Also: CTIT technical report 96–26, University of Twente, Enschede, The Netherlands.

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baten and S. Mauw, editors, *CONCUR'99 – 10th Int. Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 46–65. Springer Verlag, 1999.

[Tre02] J. Tretmans. *Testing Techniques.* University of Twente, 2002. Course notes for the course on Testing Techniques.

[Tre08] J. Tretmans. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.

[TTC] TTCN-3 test system reference architecture web page. `http://www.ttcn-3.org/index.php/about/referrence-architecture`.

[TV11] J. Tretmans and M. Verhoef. Testing the Myrianed Protocol: JTorX, torxakis, UPPAAL TRON. In *QUASIMODO Deliverable D5.10: Final report: case studies and tool integration*, pages 10–14. 2011.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[UTP] UML Test Profile on the OMG website. `http://utp.omg.org/`.

[van11] H. M. van der Bijl. *On changing models in Model-Based Testing*. PhD thesis, University of Twente, Enschede, May 2011.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In RobertM. Hierons, JonathanP. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer Berlin Heidelberg, 2008.

[vdBRT04] H. M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In *FATES 2003*, volume 2931 of *LNCS*, pages 86–100. Springer, 2004.

[vS09] Sabrina von Styp. Towards a testing theory for timed and symbolic systems. Master's thesis, RWTH Aachen, Aachen, Germany, 2009.

[vSY13] Sabrina von Styp and Liyong Yu. Symbolic model-based testing for industrial automation software. In Valeria Bertacco and Axel Legay, editors, *Haifa Verification Conference*, volume 8244 of *LNCS*, pages 78–94. Springer, 2013.

[vY07] Jeroen van Yperen. SPEX: A Simple Promela EXplorer for TorX. Master's thesis, University of Twente, November 2007.

[WDT+11] Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, Stephan Schulz, and Anthony Wiles. *An Introduction to TTCN-3*. Wiley, 2011.

[Wei09] Martin Weiglhofer. *Automated Software Conformance Testing*. PhD thesis, Graz University of Technology, 2009.

[yWo] yWorks. yEd webpage. `http://www.yworks.com/en/products_yed_about.html`.

Index

U

V

Samenvatting

Het hoofddoel van het werk, dat we in dit manuscript beschrijven, is: “Het ontwerpen van een flexibel en state-of-the-art tool voor model-gebaseerd afleiden en automatisch uitvoeren van black-box tests voor reactieve systemen. Het tool moet zowel bruikbaar zijn voor educatieve doeleinden, als ook buiten een academische context.” Van dit hoofddoel leiden we functionele en niet-functionele eisen af voor het ontwerp van het tool. De kern van het manuscript is een bespreking van het ontwerp van het tool, waarin we laten zien hoe aan de functionele eisen voldaan wordt. Daarnaast presenteren we materiaal—case studies, en antwoorden op een enquete die we gehouden hebben onder gebruikers van het tool—dat laat zien in hoeverre aan de niet-functionele eisen voldaan wordt.

We bespreken de architectuur van het tool, en beschrijven een drietal gebruiksscenarios die het tool moet kunnen ondersteunen, wil het aan de functionele eisen kunnen voldoen: “random on-line” testen, “guided on-line” testen, en het “off-line” afleiden en uitvoeren van tests. Bij het “on-line” testen vinden test-afleiding en test-uitvoering geïntegreerd plaats: een test-stap wordt pas afgeleid op het moment dat die nodig is om de test-uitvoering voortgang te laten vinden. Bij het “random on-line” testen wordt tijdens de test-afleiding een willekeurig pad door het model doorlopen. Bij het “guided on-line” testen wordt, naast het model waaruit de tests worden afgeleid, extra sturingsinformatie gebruikt, die aangeeft welke paden in het model doorlopen moeten worden bij het afleiden van de test. Bij het “off-line” afleiden en uitvoeren van tests vinden het afleiden en uitvoeren plaats als gescheiden activiteiten.

In de architectuur van ons tool onderscheiden we twee hoofdonderdelen: één voor testafleiding en één voor testuitvoering. Het hoofdonderdeel voor testafleiding leidt testprimitieven (teststappen) af uit een gegeven model en uit eventueel daarbij gegeven sturingsinformatie. Het hoofdonderdeel voor testuitvoering bevat de functionaliteit voor het koppelen van het tool aan een te testen systeem—we noemen dit de “adapter”. We bespreken het hoofdonderdeel voor testafleiding aan de hand van het bovengenoemde drietal gebruiksscenarios, en beschrijven daarnaast faciliteiten voor visualisatie, en voor het omgaan met “divergence” in modellen. Bij de beschrijving van het hoofdonderdeel voor testuivoering bespreken we drie adapter voorbeelden, waarna we deze veralgemeniseren tot een algemeen ontwerp voor een adapter. We sluiten af met een beschrijving van uitbreidingen voor het op symbolische wijze kunnen omgaan met data en tijd.

Titles in the IPA Dissertation Series since 2008

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms.* Faculty of

Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative*

Environments. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates.* Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Fac-

ulty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards*. Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor*. Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins*. Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points*. Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model*. Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development*. Faculty of Science, Mathematics and Computer Science, RU. 2013-15

C. Tankink. *Documentation and Formal Mathematics — Web Technology meets Proof Assistants*. Faculty of Science, Mathematics and Computer Science, RU. 2013-16

C. de Gouw. *Combining Monitoring with Run-time Assertion Checking*. Faculty of Mathematics and Natural Sciences, UL. 2013-17

J. van den Bos. *Gathering Evidence: Model-Driven Software Engineering in Automated Digital Forensics*. Faculty of Science, UvA. 2014-01

D. Hadziosmanovic. *The Process Matters: Cyber Security in Industrial Control Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-02

A.J.P. Jeckmans. *Cryptographically-Enhanced Privacy for Recommender Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-03

C.-P. Bezemer. *Performance Optimization of Multi-Tenant Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2014-04

T.M. Ngo. *Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs*. Faculty of

Electrical Engineering, Mathematics & Computer Science, UT. 2014-05

A.W. Laarman. *Scalable Multi-Core Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-06

J. Winter. *Coalgebraic Characterizations of Automata-Theoretic Classes*. Faculty of Science, Mathematics and Computer Science, RU. 2014-07

W. Meulemans. *Similarity Measures and Algorithms for Cartographic Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2014-08

A.F.E. Belinfante. *JTorX: Exploring Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2014-09

JTorX: Exploring Model-Based Testing

Axel Belinfante