TorX : Automated Model Based Testing

Côte de Resyste

Jan Tretmans	Ed Brinksma
University of Nijmegen	University of Twente
Dept. of Software Technology	Formal Methods and Tools Group
Faculty of Science	Faculty of EEMCS
P.O. Box 9010	P.O. Box 217
NL–6500 GL Nijmegen	NL–7500 AE Enschede
`tretmans@cs.kun.nl`	`brinksma@cs.utwente.nl`

Abstract

Systematic testing is very important for assessing and improving the quality of software systems. Yet, testing turns out to be expensive, laborious, time-consuming and error-prone. The Dutch research and development project *Côte de Resyste* worked on methods, techniques and tools for automating specification based testing using formal methods. The main achievement of the project is a test tool, baptized TorX, which integrates automatic test generation, test execution, and test analysis in an on-the-fly manner. On the one hand, TorX is based on well-defined theory, viz. the **ioco**-test theory, which has its roots in the theory of testing- and refusal-equivalences for transition systems. On the other hand, the applicability of TorX has been demonstrated by testing several academic and industrial case studies. This paper summarizes the main results of the project *Côte de Resyste*.

Keywords: model-based testing, specification-based testing, formal methods, test automation, test generation.

1 Introduction

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

Despite its importance, testing is often an under-exposed phase in the software development process. Moreover, testing has turned out to be expensive, difficult, and problematic. One source of problems is that specifications are usually imprecise, incomplete and ambiguous, so that a good basis for testing is lacking. Another source is that testing is often a manual and laborious process without effective automation, so it is error-prone and consumes many resources. The testing phase often gets jammed between moving code delivery dates and fixed custom delivery dates. Besides, research and development in testing have been rather immature. Testing methodology is mostly ad hoc and governed by heuristics.

Fortunately, this situation is gradually improving. Triggered by the quest for improved quality and imposed by increased product liability, testing is considered increasingly important and treated more seriously. Research in software testing is growing, the testing phase is more seriously planned and managed, and being a software tester is starting to be a true profession.

Côte de Resyste The research and development project *Côte de Resyste*– *Co*nformance *T*esting *of* *Re*active *Syst*ems – aimed at improving the testing process by using formal methods.

Côte de Resyste was supported by the Dutch Technology Foundation STW in the context of the "Progress" programme. The project is a cooperation between Philips Research Laboratories Eindhoven, Lucent Technologies R&D Centre Twente, Eindhoven University of Technology, and the University of Twente, while close relationships existed with CMG (now LogicaCMG) and Interpay. The 23 man-year project started in 1998 and ended in 2002.

Model Based Testing *Côte de Resyste* worked on improving the software testing process by enabling automatic testing of software systems based on formal models of these systems. In doing so, *Côte de Resyste* concentrated on *specification based*, *functional testing* of *reactive systems*.

Reactive systems are mostly technical, event-driven software systems in which stimulus / response behaviour is very important. Examples are embedded systems, communication protocols, and process control software. Administrative systems are typically not reactive systems.

Testing involves checking the correctness of a reactive system by performing experiments in a systematic and controlled way. Functional testing involves checking whether the system behaves correctly: does the system do what it should do, as opposed to, e.g., testing the performance, robustness, reliability, or user-friendlyness. Specification-based refers to the existence of a specification which exactly prescribes what the system shall do and what not. This specification is the starting point for testing. The System Under Test, referred to as the SUT, is considered a black box about which no internal details are known.

With formal, model based testing the specification is given as a model in some formal language. This formal specification is the starting point for testing the SUT.

Automated Testing Different phases can be distinguished in the testing process. During *test generation* a test suite is developed starting from a specification of the SUT. This test suite is usually expressed in an abstract way. The next step is to rewrite or implement this abstract test suite so that it can be executed. This is referred to as *test implementation*. During *test execution* the implemented test suite is executed on the SUT. Finally, the test results should be analysed, and compared with expected results: *test result analysis*.

Traditionally, test automation refers to automation of test execution, and sometimes to test analysis. A test must be devised by humans and written down in a usually low-level, test tool specific scripting language before automatic execution can start. *Côte de Resyste* aimed at automation of the whole testing process starting with test generation up to and including test result analysis. This opens the way towards completely automatic testing, where the system under test and its formal specification are the only required prerequisites.

Overview This paper outlines the main results of the *Côte de Resyste* project, including pointers for further reading. The main result of the project is a *test tool*, baptized TorX, which has, on the one hand, a well-defined and sound *theoretical basis*, and, on the other hand, high practical *applicability*. The theoretical basis is outlined in Section 2. TorX itself is described in Section 3. The applicability was evaluated by performing different case studies supplied by the companies Philips, Lucent, CMG and Interpay. They are further discussed in Section 4. Section 5 gives the main conclusions, the open issues, and hints for further research.

2 Theory

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

With formal methods, systems are specified and modelled by applying techniques from mathematics and logic. Such formal specifications and models have precise, unambiguous semantics, which enables the analysis of systems and the reasoning about them with mathematical precision and rigour. Moreover, formal languages are more easily amenable to automatic processing by means of tools. For example, verification tools exist that are able to verify fully automatically the absence of deadlock based on a formal description of the design. Until recently, formal methods were merely an academic topic, but now their use in industrial software development is increasing, in particular for telecommunication software and for safety critical systems. An example of a project where formal methods have successfully been used is the control system for the storm surge barrier in the Nieuwe Waterweg near Rotterdam [29].

In *Côte de Resyste* we have been using *labelled transition systems* as the underlying formal basis. A labelled transition system consists of states, representing the states of a system, with labelled transitions between the states. The labels on the transitions represent the observable actions of a system, such as inputs and outputs. Many formal specification languages can semantically be expressed in terms of transition systems. Of these, we used LOTOS [19], PROMELA [18], SDL [7], and FSP [21].

Testing with Formal Methods A formal specification is a precise, consistent and unambiguous basis for software design and code development as well as for testing. To define which implementations are correct with respect to a transition system specification and which are not, an *implementation relation* (or satisfaction relation) is defined. In this way it is also determined which implementations should *pass* a generated test suite, and which implementations should *fail*.

TORX is based on the **ioco**-test theory to define correctness [27, 28]. The implementation relation **ioco** has its roots in the theory of testing equivalences and preorders for transition systems [9, 6].

Formally, the definition of **ioco** is

$$i \;\mathbf{ioco}\; s \quad \Leftrightarrow_{\mathrm{def}} \quad \forall \sigma \in \mathit{Straces}(s) : \mathit{out}(\, i \;\mathbf{after}\; \sigma \,) \subseteq \mathit{out}(\, s \;\mathbf{after}\; \sigma \,)$$

Informally, this means that an implementation i (which is seen as a transition system) is **ioco**-correct with respect to a specification s (which is also a transition system), if, and only if, after all possible behaviours of the specification ($\forall \sigma \in \mathit{Straces}(s)$), any output action x produced by the implementation ($x \in \mathit{out}(\, i \;\mathbf{after}\; \sigma \,)$) can also occur as an output of the specification ($x \in \mathit{out}(\, s \;\mathbf{after}\; \sigma \,)$). In particular, this should also hold for the special action *quiescence*, which models the absence of outputs.

This formal notion of correctness is the starting point for a test generation algorithm which derives a test suite from a transition system specification to test for **ioco**-correctness. A test suite generated with this algorithm has two important, provable properties:

- *soundness*: if a test *fails* with an implementation, then this implementation is *not* **ioco**-correct;
- *exhaustiveness*: if an implementation is *not* **ioco**-correct, then there is a test in the test suite which *fails*.

Formal methods provide a rigorous and sound basis for algorithmic and automatic generation of tests. Having a precise and unambiguous specification together with a clear notion of what a correct implementation is, is a big advantage in contrast with traditional testing processes, where such a formal test basis is usually lacking.

Test Selection There is, however, also a disadvantage of the **ioco**-test derivation algorithm: for almost any realistic system an exhaustive test suite will contain infinitely many test cases, so that such a test suite can never be executed. Therefore a finite selection from the infinite exhaustive test suite is necessary. By making such a selection exhaustiveness is lost, but soundness is preserved.

Test selection is a difficult task. A simple solution is to make a random selection, and although our experiments show that this can be quite satisfactory, it is better to adopt some selection strategy or to apply selection criteria. A selection strategy should aim at detecting as many erroneous implementations as possible within a restricted period of time: it should maximize the chance of detecting an error while minimizing the cost of executing the test suite.

Part of *Côte de Resyste*'s theoretical research was devoted to test selection. Two approaches have been pursued, referred to as "test purposes approach" and "heuristics approach".

It is important to note that for test selection additional information in the test derivation process is necessary. The formal specification prescribes which behaviour is allowed and which is not. It does not give information about which behaviour is more important, or which behaviours are more likely to contain errors. Such information is important for test selection, but it cannot be found in the formal specification, so it must come from elsewhere.

Test Purposes In the "test purposes approach" it is the user (person performing the tests) who supplies information about which behaviours are important or are likely to contain errors. The user does this by specifying *test purposes*: behaviours which (s)he wants to observe and test to be sure that they are correctly implemented. This approach is also referred to as "user guided" test selection.

The "test purposes" approach was formally elaborated [32]. A framework has been developed in which test purposes are formalized as *observation objectives*, which can be *hit* or *missed* when executing a test. An observation objective is orthogonal to correctness, and it can be very specific, e.g., one specific trace, or it can be very general, e.g., all behaviours in which inputs are only supplied when the SUT is quiescent.

This approach was worked out for the **ioco**-test derivation algorithm: an observation objective gives the extra information to guide the test derivation in the direction of a test case which can *hit* the observation objective. This new algorithm was proved to be *e-exhaustive* and *e-sound*; for details see [32].

Heuristics An alternative approach to test selection is to provide the extra information for test selection in the form of predefined strategies based on heuristics [12]. These heuristics are based on assumptions about the behaviour of the system under test. Three heuristic principles have been elaborated referred to as "length heuristic" (testing a finite prefix of an infinite trace is assumed to be sufficient), "cycling heuristic" (testing a finite number of iterations of a transition-system cycle is assumed to be sufficient), and "reduction heuristic" (if a state has infinitely many outgoing transitions of the same shape then testing a finite number of them is sufficient). These heuristics have been formalized by defining a *distance function* on behaviour traces. The maximum distance between the traces in a test suite and those not in that suite then leads to a definition of test suite *coverage*; for details see [12].

3 A Tool

TorX: A Tool for Formal Testing One of the main achievements of *Côte de Resyste* is the prototype test tool TorX. TorX provides automatic test generation, test implementation, test execution and analysis in an *on-the-fly* manner [4, 26]. TorX implements the **ioco**-test derivation algorithm to derive tests from formal, transition system-based specifications. This includes test selection by means of test purposes, see Section 2. The specifications can be expressed in the formal languages Lotos, Promela or Fsp, or directly as a transition system in the Aldebaran-format [13]. The first two languages were mainly used in the case studies (see Section 4); the latter two are very useful for educational purposes.

In TorX, automatic test generation and test execution are not done in separate phases but they are integrated, i.e. there is no complete test suite generated that is subsequently executed. During test execution, tests are derived on-the-fly (or lazily, cf. lazy evaluation of functional programming languages). For each test step, TorX computes only the *test primitives* from the formal specification which are needed in that step: the stimuli that can be given, and the observations that are expected. It then performs the test step: it decides between stimulating and observing, and then either chooses a stimulus and sends it to the implementation, or it acquires an observation from the implementation, and checks whether it was expected (and reports an error if not). After sending the stimulus or checking the observation (and finding no error in it), it computes the test primitives for the next test step, performs the next test step, etc.

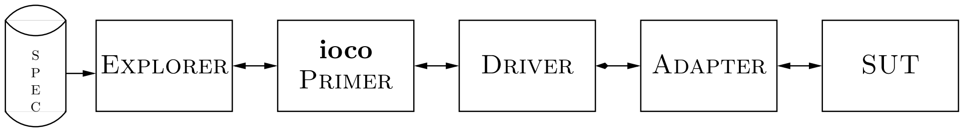

Figure 1: On-the-fly test generation and execution.

This repeated derivation and execution of test steps can be done fully automatically without any user intervention, as described above (this is very useful for case studies), but also semi-automatically under control of the user (this is useful for demonstrations, and for studying particular scenarios in detail). For fully automatic derivation and execution, the user only has to provide the maximum number of test steps that should be performed. During user-controlled derivation and execution, the test primitives that have been computed are presented to the user, who can decide between stimulating and observing, and, if stimulating, can choose the particular stimulus that is to be sent to the implementation.

A test run is collected in a log, containing all the test steps executed (both in abstract form, as they appear in the specification, and in concrete form, as the bits and bytes communicated with the SUT). The test log is visualized on-the-fly as a *message sequence chart*. A recorded test log can later be re-executed.

The Architecture of TorX The main characteristics of TorX are its flexibility and openness. Flexibility is obtained by requiring a modular architecture with well-defined interfaces between the components – this allows easy replacement of a component by an improved or modified version (e.g., one that supports another specification language or test generation algorithm). Openness is achieved by using standard interfaces to link the components of the tool environment – this enables integration of third-party components that implement these interfaces.

The TorX architecture consists of the following basic components that are mandatory in any use of TorX, see Figure 1: Explorer, Primer, Driver, and Adapter. The following components are optional and can be "plugged-in" when a particular feature is needed: Combinator, Partitioner, IOchooser, and Instantiator. The well-defined interfaces allow this "plugging in". Figure 2 depicts how all these components can be linked for on-the-fly test derivation and execution. The SUT is the system under test. This role can also be played by a simulated specification.

The Explorer is a specification language specific component that offers functions (to the Primer) to explore the state-transition graph of a specification. TorX contains Explorers for Lotos (using the Cæsar/Aldebaran Development Package [14]), Promela (based on Spin [31]), Fsp (using the Ltsa analyser [21]), automata (using Aldebaran), and any other specification language for which an Open/Cæsar interface exists [14].

The Primer uses the functions of the Explorer to implement the test derivation algorithm that generates the test primitives from the state-transition graph.

The Driver is the central component of the tool architecture. It controls the testing process by deciding whether to stimulate the SUT, or to make an observation and check it. The Driver can be run in two modes (see above): a manual mode, in which the user is in full control, and an automatic mode, in which the Driver makes all necessary choices randomly (or guided by probabilities; see below).

The Adapter is the test application specific component that provides the connection with the SUT. It is responsible for sending inputs to, and receiving outputs from the SUT on request of the Driver, and for encoding and decoding of abstract actions from the Driver into the concrete bits and bytes for the SUT, and vice versa. This includes the mapping of time-outs onto quiescence actions. This clearly makes the Adapter dependent on both the specification (version, language), and the SUT.

The optional Combinator is used to combine test primitives from multiple sources (like Primers or Combinators themselves – they can be cascaded). In particular, it is used to combine the test primitives of a specification with those derived from a test purpose. Test purposes

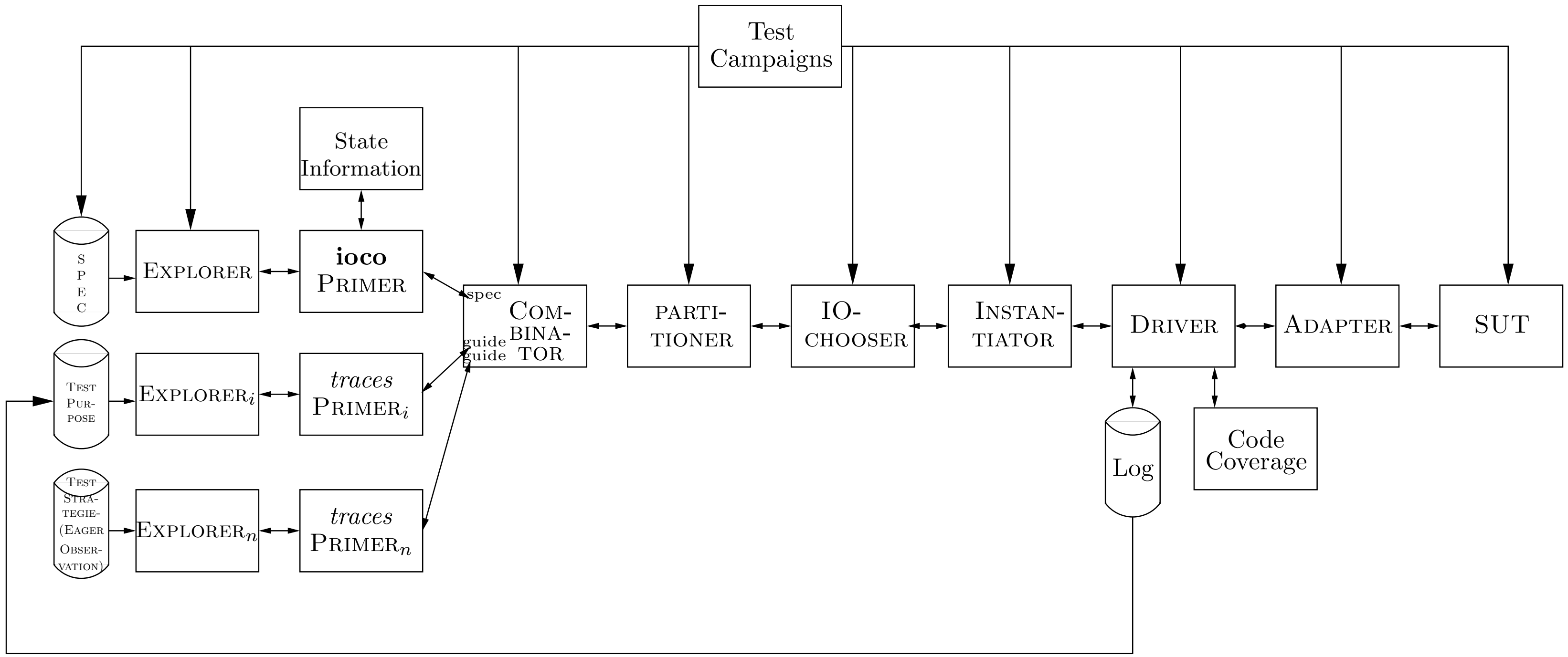

Figure 2: On-the-fly test generation and execution with test purposes, campaigns, variable instantiation, and probabilistic choices.

can be represented in the same languages as the specification and then the same EXPLORERs are used. Alternatively, they can be expressed and processed using a special test-purpose language JARARACA.

The optional PARTITIONER is used to steer the on-the-fly derivation process. Normally, when we want to stimulate the SUT, we choose randomly with equal distribution from the set of possible inputs. With the PARTITIONER we guide this selection by dividing the possible input test primitives into partitions to which weights (probabilities) are assigned. These weights are taken into account when an input is chosen. PARTITIONERS can be cascaded to partition input actions according to multiple criteria.

Also the optional IOCHOOSER is used to steer the on-the-fly derivation process. Normally, we choose randomly with an equal distribution between stimulating and observing. With the IOCHOOSER we guide this choice by attaching weights (probabilities) to stimulating and observing. These weights are taken into account when a choice between stimulating and observing is made. IOCHOOSER and PARTITIONER implement the ideas presented in [11].

The optional INSTANTIATOR is used to instantiate free variables in the test primitives (stimuli) computed by the PRIMER. This is necessary because the ADAPTER is not able to encode stimuli that contain free variables.

Interfaces of TORX To support the openness of the TORX architecture, we connect components by pipes over which textual commands and responses are exchanged – these textual interfaces make it simple to debug and test individual components, to experiment using (Unix-style) filters to massage the information exchanged, and even to split the tool over several machines. The textual interfaces used between the TORX components all have the form of a remote procedure call: a component issues a request to another component after which this component replies. In the TORX architecture the components are connected pairwise; a TORX configuration forms a tree of components with the DRIVER as root. For each pair the component closest to the root of the tree (the parent) will take the initiative to issue requests, and the other component (the child) will only reply to them (but, in order to do so, it may issue request(s) to its own child(ren), and use their responses to compute its own response to its parent). In addition to the textual interfaces, some existing standard interfaces are supported, like the OPEN/CÆSAR interface [14] and the GCI interface [5].

Test Campaigns During several case studies it turned out desirable to have different test runs executed after each other without user interaction. To make this possible, *test campaigns* were developed. Test campaigns make it possible to specify, schedule and manage several test runs, all with different TORX configurations, different parameters, different input distributions, and even different specifications or implementations. Moreover, the results of all these test runs are systematically archived. The implementation of test campaigns consists of a layer on top of TORX.

4 Applications

The Conference Protocol The first case study within *Côte de Resyste* was the *Conference Protocol*. It is a simple, yet realistic chatbox protocol that runs on top of the internet protocol UDP. Conference Protocol Entities (CPE's) were tested with TORX based on specifications in the formal languages PROMELA and LOTOS. As implementations we used a set of 28 different CPE's, implemented in C, of which one was (assumed-to-be) correct, 25 were erroneous mutants obtained by introducing single errors in the correct one, and 2 were modified but **ioco**-correct implementations.

The aims of the *Conference Protocol* case study were (*i*) to evaluate the TORX test tool prototype, and to assess its error-detection power by mutation analysis on the 28 CPE implementations; (*ii*) to compare TORX with other specification based test tools; and (*iii*) to use it as benchmark for future versions of TORX, and to experiment with new functionality in TORX.

The test method for testing a CPE was black-box conformance testing via a distributed test method, where the tester – TorX – played the roles of the local CPE user and two remote conference partners. The tester has access to the CPE under test through three PCOs (Points of Control and Observation). In the role of local CPE user, the tester has direct access at the upper service access point of the CPE, whereas in the role of the remote conference partners it has access via the underlying UDP layer. The remote users were not really 'remote': the tests were executed on a single computer, on which both the testing tool and the CPE under test were running.

From the set of 28 CPE implementations, TorX was able to successfully detect all erroneous ones. At most 500 test events were needed to detect the errors using random test selection. With the correct implementations, long test runs consisting of more than 450,000 test events were generated and executed completely automatically without detecting any error [4].

Apart from evaluating TorX, the Conference Protocol was used as a bench-marking experiment to compare TorX with some other specification based test generation tools. Firstly, an SDL specification of the Conference Protocol was developed from which 13 test cases were generated using the SDL test tool Tau. These 13 test case were executed on the CPE's, but they were not able to detect 6 erroneous mutants [4]. Secondly, for the FSM-based test generation tool Phact/*Conformance Kit*, an EFSM (Extended Finite State Machine) specification was developed, from which 82 test cases were derived. Three erroneous implementations passed this test suite [17]. This confirmed our hypothesis that FSM-based software testing is inferior to transition system-based testing. Thirdly, for the test tool Tgv, the Lotos specification was used again. Tgv is also based on the **ioco**-theory, and like TorX, was able to detect all erroneous implementations [10]. Beside these experimental comparisons, a theoretical comparison between these different test generation methods was made with analogous results [15].

The Conference Protocol, being small yet realistic, turned out to be a very suitable case study for TorX. It provided valuable feedback for improving TorX, and it was useful for bench-marking, for doing experiments with new extensions, for demonstration purposes, and for use in courses. To allow others to use the Conference Protocol as a bench-mark for their testing tools, a web site was constructed containing documentation, all formal specifications, and our implementations [22].

"Rekeningrijden" For Interpay B.V. *Côte de Resyste* performed a case study to evaluate the applicability of formal testing techniques. The study consisted of testing a part of the Payment Box, which is part of the once advocated Highway Tolling System – in Dutch "Rekeningrijden". This system automatically charges fees from vehicle drivers who pass a toll gate on a highway. The fee is paid electronically by means of exchanging digital certificates between the Payment Box in the toll gate and an electronic purse on a smart card in the passing vehicle. When a vehicle passes the toll gate, the system should debit the purse and register a balance increment at the Payment Box. Because many vehicles can pass a toll gate simultaneously and since the vehicles travel at high and different speeds, the number of parallel transactions in progress can be large. Furthermore, for security reasons, the messages exchanged for an electronic payment transaction are encrypted. These issues – speed, parallelism and encryption – contribute to the complexity of testing. The object of testing was the Payment Box side of the protocol between Payment Box and smart card [30].

The Payment Box had been tested by Interpay in a traditional way. Tests had been manually developed and automatically executed using a dedicated test execution environment. The latter was necessary to meet the speed and encryption requirements.

Before starting, we developed a generic step-wise approach in which all the activities for formal testing are embedded [30]. Subsequently, the case study was carried out following this approach. First, we studied the IUT (Implementation Under test) and wrote formal specifications in Lotos and Promela starting from the informal documents. While writing and validating this formal specification (by model checking with Spin [18]) we detected an important design error. Before continuing this error was repaired. In the second step, we studied the test tools with respect to their ability to test the IUT and their means to interface with the SUT. We reused part of the existing test environment for traditional testing. Third, the results of the first and second step were combined, as basis for the development of the test environment containing both the test tools and the IUT. Most time was spent in this phase. It turned out that we were not able to interact directly with the Payment Box, due to the encryption involved in electronic transactions. Furthermore, we had to deal with the (real-) time requirements during testing. This led to significant effort in implementing the application specific tool component – the ADAPTER; see Section 3 – and in extension of the IUT specification to contain the test context. In the fourth step, several test runs, with length up to 50,000 test events, were automatically generated and executed. These runs were specified and scheduled using test campaigns; see Section 3. During test execution, one error was detected, which is still under study by Interpay.

The main result with respect to the Payment Box is that two defects were found. The most important one was a design error which was not detected during testing but during formal specification and subsequent validation.

With respect to TORX and the *Côte de Resyste* methodology we have the following conclusions:

- There is insufficient support, both in theory and in tools, for testing applications with real-time behaviour. In particular, the difference between quiescence (see Section 2) and time-out is confusing and not well-understood.
- The performance of TORX' test derivation needs to be improved: TORX was not always able to calculate the next test primitives before the Payment Box gave a time-out. The PROMELA specification performed much better in this respect than the LOTOS one.
- Our hypothesis that TORX can easily deal with parallelism was confirmed. Having many cars in parallel was conceptually no problem, although it sometimes gave problems with respect to performance; see above.
- Implementing a test execution environment is a laborious process, although not harder than for manual testing. More generic approaches for implementation of test environments (i.e., ADAPTERS) are needed.
- Detecting an error is one thing; analysing and repairing it is another: more tool support for test result analysis is needed.
- TORX is easily distributed over multiple platforms: the Payment Box was running on VXWORKS, the ADAPTER on WINDOWS-NT, and the rest of TORX on LINUX.
- The concept of *test campaigns* was mainly developed for, and during this case study. It proved to be very valuable.

Altogether, we conclude that the *Côte de Resyste* approach is not yet mature enough to cope with applications like the Payment Box, which is mainly due to timing – real-time and performance requirements. But the automated test approach turned out to be very flexible, reliable, and fast: large numbers of long tests were easily derived and executed. Certainly, formal specification and validation should be used for the type of protocols as used in the Payment Box. From a research point of view, the case study was successful, and a step ahead in formal testing of realistic systems. Many new ideas and research items were identified and TORX was improved and extended.

The EasyLink Protocol Philips' *EasyLink* Protocol concerns the communication between a video recorder and a television set. The TV-side of the preset-download feature of this protocol was tested with TORX based on a PROMELA model. Functions like initiating a preset-download, stopping downloading at the end or somewhere in the middle, and shuffling the presets with the TV remote control while downloading, were tested; see [3] for the details of this test effort.

For the test environment, the messages between VCR and TV were caught using a specialized probe, which also allowed to insert messages. This probe communicated with a PC, which then communicated with an HP-workstation on which the main parts of TORX were running.

The results of this study were promising: some (non-fatal) faults were detected which had slipped through the conventional testing procedures. Moreover, we concluded that automatic specification-based testing of this kind of product is feasible and beneficial.

And Further With CMG we tested a componenent of their *Cell-Broadcast-Centre*. Using a Lotos specification (28 pp.), their existing test environment, and an Adapter generated from an IDL description of the component interfaces, many tests were performed. But these tests with TorX did not reveal any errors which had not been detected with conventional testing, although TorX reached a slightly higher code coverage [8].

Lucent R&D Centre Twente used TorX to test the implementation of an access network protocol: Lucent's implementation of the ETSI standard for the V5.1 PSTN Access Network Protocol. Since there was no clear specification available to serve as the basis for the formal model the code was more or less re-engineered. Testing with TorX was feasible but did not discover any faults since the model was derived from the code itself [16].

In two other case studies with CMG we investigated the use of TorX for testing the control software of the *Stormvloedkering Oosterschelde* [2], and we studied the combination of TorX with their TestFrame method [23].

Moreover, the design of TorX inspired Philips in their development of a new hardware-design tester.

Conclusions Taken together, the main outcomes of the case studies are:

- Formal models serve as a precise arbiter for testing, so that only valid tests are generated, i.e., tests that test what should be tested.
- Very long tests, depending on the case study from $50,000$ up to $500,000$ test events, were automatically generated and executed.
- In some of the case studies faults were detected which had slipped through the conventional testing procedures. Strong points of TorX are that it can easily cope with a high degree of parallelism and that it can detect errors which only occur after long sequences of events.
- In cases where a comparison with traditional test methods was made, TorX performed "at least as good as" traditional testing.
- Building a test environment for executing the generated tests is laborious, but does not differ from traditional test execution automation. Traditional test environments can be reused for formal testing. In most case studies, making the models was relatively easy compared with building the test environment.
- The most important errors are usually not found by testing, but during development of the formal model for testing, e.g., when this model is analysed using model checking.

5 Concluding Remarks

Conclusion The goal of *Côte de Resyste* was to develop theory, tools and applications for automatic specification based testing using formal methods. To a large extent this goal has been achieved. The **ioco**-test theory provides a well-defined and rigorous basis for formal testing with proved test derivation algorithms. The prototype test tool TorX can completely automatically derive tests from formal specifications, execute them, and analyse the results. The successful application of TorX to different case studies showed the feasibility of the methodology, and the improvements of the testing process which were gained in terms of more, longer and provably correct tests.

Altogether, these results lead us to believe that it is advantageous to perform automatic testing based on a formal model of the system under test. The extra effort required for developing the necessary formal model is more than compensated by faster, cheaper, more effective, and more flexible testing.

The use of formal methods can improve the testing process, and formal testing can improve software development. An important benefit is not in testing itself, but in the formalization and validation process preceding the formal testing process. Then the most important errors, such as design errors, are detected. In the other direction, formal testing can stimulate the use of formal methods, by exploiting the perceived benefits during testing.

The Future TorX is only a prototype, and the case studies have clearly shown that it cannot cope with all kinds of testing in all circumstances. Moreover, there are still a number of important open testing problems. We mention some of them:

- Although important improvements have been made in test selection, it is still one of the most important research questions: how can the completeness and coverage of an automatically generated test suite be expressed, measured, computed, and, ultimately, controlled. Even more intriguing is the question how test suite coverage can be related to a measure of product quality. After all, product quality is the only actual reason to perform testing.
- Testing real-time requirements is an important issue, in particular in embedded systems. Neither the theory nor TorX can currently deal with them.
- Large data domains lead to state-explosion. Symbolic ways of representing and manipulating data are required.
- Systematic test data selection is currently not done, but is needed.
- Sometimes an abstract action in the specification is implemented as a sequence of less abstract actions in the implementation. This is called *action refinement*. Both theoretical and tool support are needed for this.
- Several case studies have shown that the performance of TorX should be improved, in particular with respect to the on-the-fly calculation of test primitives.
- Implementing a test environment, in particular the Adapter, is laborious. More support is needed, and some case studies showed that this is feasible, e.g., by generating the Adapter from an interface description in IDL. Alternatives may be ASN.1 or XML.
- Support for test log analysis can be improved, in particular, localization of an error in the implementation is currently not at all supported.
- Several cases concluded that the formal languages that we currently use are not satisfactory. A language that combines specification of behaviour and data both with formal semantics, that is user-friendly not only for formalists, for which there is sufficient tool support including seamless integration of testing and verification tools, is desirable.
- TorX tests functional properties. Extension with non-functional quality characteristics as robustness, performance, usability, reliability, ..., can be considered in the future.
- TorX was developed for reactive systems. Possible extensions may consider other kind of software systems, e.g., administrative systems.

The work on TorX is continued in several *Côte de Resyste* successor projects: action refinement is investigated in the research project Atomyste [1], real-time and data extensions for TorX are studied in the project Stress [24], testing of functions based on relations between input and output is investigated [20], and the application of TorX to testing *wafer scanners*, including the necessary hybrid, real-time, data, and compostionality extensions, is investigated in Tangram [25].

Availability TorX is freely available for research purposes [26].

Acknowledgements

Many people contributed to the success of *Côte de Resyste*. We thank Lex Heerink and Ron Koymans from Philips Research Laboratories Eindhoven, Arjan de Heer from Lucent Technologies R&D Centre Twente, Erik Kwast and Henri Dol from KPN Research, Loe Feijs, Sjouke Mauw and Nicolae Goga from the Eindhoven University of Technology, Axel Belinfante, Jan Feenstra and René de Vries from the University of Twente, and Peter Christian, Robert Spee, and Wouter Geurts from CMG, for their active participation in the project. CMG and Interpay, in particular Cornel van Mastrigt and Rommert Jorritsma, are thanked for the support they gave in performing the case studies. The financial support from STW and from the NWO Van Gogh programme is acknowledged.

References

[1] ATOMYSTE – Atom Splitting in Embedded Systems Testing. http://fmt.cs.utwente.nl/projects/ATOMYSTE-html/.

[2] A. Belinfante. Timed Testing with TorX: The Oosterschelde Storm Surge Barrier. In M Gijsen, editor, *Handout* 8^e *Nederlandse Testdag"*, Rotterdam, The Netherlands, November, 20 2002. CMG.

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

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

[5] F. Brady and R.M. Barker. Infrastructural Tools for Information Technology and Telecommunications Conformance Testing, INTOOL/GCI, Generic Compiler/Interpreter interface (GCI) Interface Specification, Version 2.2, 1996. INTOOL document number GCI/NPL038v2.

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

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

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

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

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

[11] L.M.G. Feijs, N. Goga, and S. Mauw. Probabilities in the TORX Test Derivation Algorithm. In S. Graf, C. Jard, and Y. Lahav, editors, *SAM 2000 –* 2^{nd} *Workshop on SDL and MSC*, pages 173–188. VERIMAG, IRISA, SDL Forum Society, June 2000.

[12] L.M.G. Feijs, N. Goga, S. Mauw, and J. Tretmans. Test Selection, Trace Distance and Heuristics. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems XIV*, pages 267–282. Kluwer Academic Publishers, 2002.

[13] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP – Cæsar/Aldebaran Development Package: A Protocol Validation and Verification Toolbox. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification CAV'96.* Lecture Notes in Computer Science 1102, Springer-Verlag, 1996.

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

[15] N. Goga. Comparing TorX, Autolink, TGV and UIO Test Algorithms. In R. Reed and J. Reed, editors, *SDL 2001: Meeting UML*, volume 2078 of *Lecture Notes in Computer Science*, pages 379–402. Springer-Verlag, 2001.

[16] A. de Heer. Automated Testing of V5.1 PSTN – A Case Study into the Applicability of Automated Model Based Testing. Internal report, Lucent Technologies R&D Centre Twente, Enschede, The Netherlands, 2001.

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

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

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

[20] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic Automated Software Testing. In R. Peña, editor, *IFL 2002 – Implementation of Functional Programming Languages*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer-Verlag, 2003.

[21] J. Magee and J. Kramer. *Concurrency: Sate Models & Java Programs*. Wiley, 1999.

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

[23] R. Spee. Combining TorX and TestFrame – Theory Meets Practice. In M Gijsen, editor, *Handout* 8^e *Nederlandse Testdag"*, Rotterdam, The Netherlands, November, 20 2002. CMG.

[24] STRESS – Systematic Testing of Real-Time Embedded Software Systems. `http://fmt.cs.utwente.nl/projects/STRESS-html/`.

[25] TANGRAM – Model Based Testing. `http://www.embeddedsystems.nl/`.

[26] TORX – Test Tool Information. `http://fmt.cs.utwente.nl/tools/torx`.

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

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

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

[30] R.G. de Vries, A. Belinfante, and J. Feenstra. Automated Testing in Practice: The Highway Tolling System. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems XIV*, pages 219–234. Kluwer Academic Publishers, 2002.

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

[32] R.G. de Vries and J. Tretmans. Towards Formal Test Purposes. In E. Brinksma and J. Tretmans, editors, *Formal Approaches to Testing of Software – FATES'01*, number NS-01-4 in BRICS Notes Series, pages 61–76, University of Aarhus, Denmark, 2001. BRICS.