Chapter 1

AUTOMATED TESTING IN PRACTICE: THE HIGHWAY TOLLING SYSTEM

René G. de Vries, Axel Belinfante and Jan Feenstra
University of Twente – Formal Methods and Tools group
Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
{rdevries, belinfan, feenstra}@cs.utwente.nl

Abstract In this paper we study the application of automated test derivation and execution based on formal specifications. The object of testing is the Payment Box (PB) of the Highway Tolling System, a device which handles electronic payments. Challenges of testing the PB are the transaction speed, parallelism and encryption. We describe a methodology for automated testing and apply this methodology to test the PB. We conclude that automation of the test process is feasible and beneficial, and evaluate our techniques, theory and tools for automated testing.

1 INTRODUCTION

In this paper we study the deployment of methods, techniques and tools which support specification based testing of reactive software systems. Systematic testing of the functionality of reactive systems is crucial in order to check whether they operate correctly. Our research goal is to develop methods and tools to support and, whenever possible, to automate the *conformance testing* process of reactive systems. Conformance testing involves assessing the correctness of a system with respect to its functional specification by means of experimentation. For it we need tools for algorithmic derivation of test suites from specifications and tools for automatic test implementation, execution and analysis.

*This research was partly supported by Interpay Nederland B.V. and by the Dutch Technology Foundation STW under project STW TIF.4111: *Côte de Resyste* – COnformance TEsting of REactive SYSTEms.

Our challenge is to develop test tools based on a well-defined theory which are well suited for practical applications. Consequently, it is important that the developed tools are applied to realistic case studies from industry in order to validate their practical usefulness. In this paper we report on testing a complex industrial device, for which we have not only generated but also executed the tests. We emphasize on the process of setting up and performing automated testing and evaluate how succesful the tools and techniques are.

Automated testing. Test derivation and test execution become increasingly laborious activities, while the complexity of reactive systems grows. Automation of both processes seems to be a solution to reduce the labour. Besides, automated testing is flexible and enables automatic regression testing when a system is modified.

Given the specification of a system, we derive test experiments and execute them on the *Implementation Under Test* (IUT). For automated derivation we need a formal specification of the IUT and formal test theory. Several theories for automated derivation from specifications are known, like [6; 9]. In this paper we adopt the test theory of [9] and use the test tool TorX [1], which exploits this theory.

TorX is also able to execute tests automatically. We use TorX in the *on-the-fly* testing mode. During on-the-fly testing, test derivation and execution occur simultaneously. Instead of deriving a complete test case (one test scenario of a test suite), the test derivation process derives *test primitives* from a specification. These test primitives are actions that are immediately executed in the test run. In this way only the necessary part of a test case is considered, which reduces the amount of computation needed. An important remark is that all computations for one-the-fly testing have to be done at run-time. This can be critical when we need to satisfy an IUT's real-time requirements. An elaborated discussion of test automation based on formal methods and a description of TorX, its architecture, testing modes and configurations can be found in [1].

Highway Tolling System. As case study for the assessment of our approach we will evaluate the testing of the *Payment box* (PB), which is a part of the *Highway Tolling System* (HTS). The Dutch government considers to charge toll on highways as an instrument to reduce the increasing traffic jams. For this purpose, the Highway Tolling System is developed. This system automatically charges fees from vehicle drivers who pass a toll gate on a highway. The fee is paid by an electronic payment which consists of exchanging digital certificates between the PB at the toll gate and an electronic purse on a smart card in the

vehicle. When a vehicle passes the toll gate, the system should debit the purse and register a balance increment at the PB.

Interpay B.V., a company owned by the Dutch banks that exploits the electronic purse smart cards, needs to test the system to ensure that the PB processes the money transactions correctly. Because many vehicles can pass a toll gate simultaneously and 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 the generation, execution and analysis of test experiments to test the PB.

Automated test derivation can make the test generation process faster and more reliable. Test generation is usually done manually by designing and writing test scripts. This is a very laborious and error-prone process, especially in systems with a high level of concurrency. Interpay B.V. tested the PB using traditional test design techniques, although there was automation involved. In order to meet the speed requirements they executed their tests in a dedicated automated execution environment. However, they desire more support for their test process.

Overview. In this paper we report on the testing of the PB using the test tool TorX. We describe the methodology of this approach in Section 2 and report on its application in Section 3. Subsequently, in Section 4 we describe the difficulties and challenges that we encountered and propose some (partial) solutions. In Section 5 we describe the results and observations we made and we finish with some concluding remarks.

2 METHODOLOGY

In this section we describe on an abstract level the process of how to automatically test a system. In the next section we will apply this process to test the PB. Because we test on-the-fly our process does not have the explicit test design phase which is present in batch-oriented testing. Although the process is presented sequentially, in practice it turns out that the process is iterative. We distinguish four phases: preparing the IUT, preparing the test tools, preparing the test environment and performing test execution. We describe them in more detail:

1. *IUT study.* In this phase, firstly we study the system that we are going to test. We investigate all potential interfaces from which we can stimulate and observe the IUT, and study the IUT's observable behaviour. This behaviour may have many features which we do not want to test. So we restrict ourself and identify the behaviour that we are

interested in. Secondly, we formalize the behaviour by specifying the system by means of *Formal Description Techniques* (FDTs), e.g. LOTOS [5] or PROMELA [4]. Since at this stage (in general), all available system descriptions are informal, we will have to resolve all impreciseness, incompleteness, ambiguities and inconsistencies from which informal specifications suffer. During the development of the formal specification we use automated validation tools to simulate, verify properties of, and model check the formalized behaviour. This gives us insight in obscurities and in anomalous behaviour that was introduced during the functional design phase. Furthermore, these tools support the design of *test purposes*, i.e. means to guide the test derivation.

2. *Tools.* In this phase we determine which tools we are going to use. We investigate what the underlying theory of the tools is, which shall be the basis of interpretation of the test results. Furthermore we investigate the openness of the tools, i.e. the interfaces that the tools have to connect with an IUT.

3. *Test environment.* The test environment is the setup of the test architecture and the application specific test tool components. The *test architecture* describes the relevant aspects of how the IUT is embedded into other systems during the testing process, and how the tester interacts with the IUT. In general we cannot connect the test tool directly with the IUT, but have to use additional devices/components to interface the test tool with the IUT. This phase is decomposed into the following four activities:
Test architecture design. A precise knowledge of the test architecture is required for the test derivation. Usually the tester does not communicate directly but indirectly with the IUT, via a so called *test context*. The IUT interacts at the *implementation access points* (IAPs) which the tester controls via the test context at the *points of control of observation* (PCOs). Consequently, the test primitives should be derived with respect to the PCOs. The IUT together with its test context is called the *System Under Test* (SUT). The tests are derived from a specification of the SUT, instead of the IUT. The components that are necessary to interface the test tool with the IUT and their functionality have to be identified. These components range from existing generic ones like network services to application specific ones, e.g. for the en- and decoding of messages exchanged between tester and IUT.
Test architecture implementation. In this phase all interfacing components will have to be designed and developed. This is a software engineering task.
System under Test specification. The test context has to be taken into consideration when we derive tests. So, the formal specification of the

IUT is extended into a formal specification of the SUT.
Testing the test architecture. We have to check the correctness of our automated test environment. In this phase we debug the text context components and adjust the specification from which we derive tests. This activity is iterative.

4. *Test execution.* In this phase we design *campaigns*, instantiated test environment setups with test goals and strategies. We execute these campaigns and analyse the results.

3 TESTING THE PAYMENT BOX

Here we tell how we tested the PB using the process presented above.

3.1 IUT STUDY

This phase consists of a global IUT study and the construction of a formal specification of the system using an FDT.

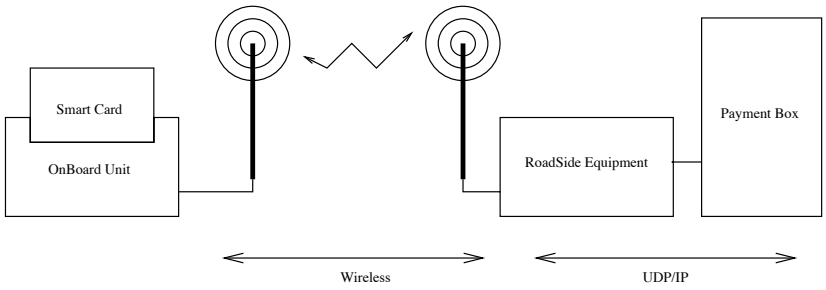

Figure 1.1 Highway Tolling System

Study IUT. The basic setup of the system is shown in Figure 1.1. A smartcard containing an electronic purse is inserted into the *OnBoard Unit* device (OBU) in the vehicle. The OBU communicates wirelessly with the *Roadside Equipment*. The Roadside Equipment is the base station in the wireless network that offers a connectionless communication service. The Roadside Equipment itself is connected to the PB by UDP/IP. During a *transaction*, i.e. an electronic payment, several messages are exchanged between the PB and the OBU, following the *PbObu* protocol rules. The protocol is simple. A payment consists of the exchange of a few messages, and there are no message recovery mechanisms. In case of message loss, the message exchange will stop, acknowledged by an error message. The subject of testing is the PB. We want to check the correctness of the electronic transaction process-

ing between multiple smart cards and the PB. So, we will attempt to connect the tester to the PB at the UDP/IP interface and assess the PB for conformance on the PbObu protocol level.

Formal model of IUT. The informal specification of the PB consists of about 50 pages of text. Since the PbObu protocol is simple it was not laborious to make a formal specification of this protocol. The PbObu protocol has been specified in LOTOS and in PROMELA. Every transaction is handled by a separate process instantiation. We used the tools CÆSAR [3] and SPIN [4] to simulate and validate the protocol.

3.2 TEST TOOLS

In this phase we determine the tools that we are going to use, investigate the underlying theory, and identify the facilities offered by the tools to interface with the IUT. In this case study we use TORX.

TorX Theory. TORX is based on the test theory for input output transition systems. In addition to input and output actions, TORX distinguishes the special action δ. The δ action, called *quiescence*, models the absence of output of a system. This absence of output can be explicitly observed by TORX (implemented by a time-out). The notion of correctness between specification and implementation is defined by the *implementation relation* **ioco**. Intuitively, an IUT is considered correct if after a sequence of input, quiescence and output actions, the observed output (including δ) of the IUT is predicted by the specification.

TorX Architecture. The main characteristics of TORX are its flexibility and openness. Flexibility is obtained by using a modular architecture with well-defined interfaces between the components. Openness is achieved by connecting components by pipes over which textual commands and responses are exchanged. The textual interfaces make it simple to debug and test individual components, to experiment using (Unix style) filters to massage the information exchanged, and to distribute the tool over several machines. When needed to link legacy components in the tool environment, we use existing interfaces.

The TORX architecture (in on-the-fly testing configuration) is depicted in Figure 1.2. The EXPLORER is a specification language-specific component that offers functions (to the PRIMER) to explore the transition-graph of a specification. The PRIMER uses these functions to implement the test derivation algorithm that generates the test primitives from the transition-graph. The optional PARTITIONER is used to steer the on-the-fly derivation process. Normally, when we want to stimulate the SUT,

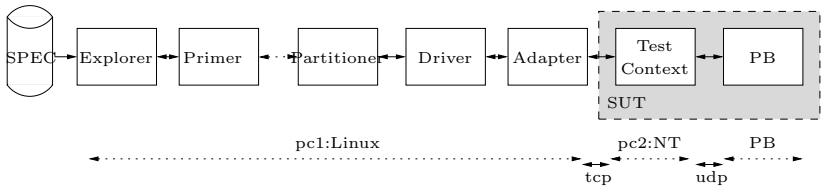

Figure 1.2 TorX tool architecture

we choose from the set of inputs randomly with a fixed distribution. 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. Similar ideas on probabilities are published in [2]. The Driver is the central component of the tool architecture. It controls the testing process, by deciding whether to stimulate or to observe and check the observation. The Driver can be run in two modes: a manual mode, in which the user is in full control, and an automatic mode, in which the Driver makes all necessary choices randomly. 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, including the mapping of time-outs onto quiescent actions. This clearly makes the Adapter dependent on both the specification (version, language), and the SUT.

3.3 TEST ENVIRONMENT

In this phase we identify the test architecture and implement and test the test environment.

Test architecture design. The PB communicates via UDP/IP, and cannot be approached otherwise. One of the underlying assumptions of testing is that the tester can synthesize every input stimulus for the IUT and can interpret every possible output. This assumption does not hold here. Due to security imposed on electronic transactions, the exchanged messages contain encrypted fields. We can not synthesize nor interpret these fields, since the encryption keys and encryption methods are classified and thus not available. We can indirectly observe and stimulate the PB using the *ObuSim*, a tool running on a Windows NT workstation containing smart cards that host the handling of encryption and decryp-

tion. We extend the ObuSim with a (TCP/IP) interface to control the ObuSim using textual commands. We stimulate and observe the IUT now indirectly. To stimulate, we request the ObuSim to synthesize a message (for which it uses the smart cards) and to send it to the PB. Messages from the PB are received by the ObuSim, decrypted using the smart cards, and the interpretation results are forwarded to the tester.

Exchanged messages and their encryption are mutually related within one transaction, i.e. the format of a message depends on preceding exchanged messages. As a result we can only synthesize, and thus only send, messages for the PB in a particular order. This restriction limits the possibility to exercise the PB behaviour. For example, we can not test for robustness (send arbitrary messages), although it is part of the system's conformance requirements.

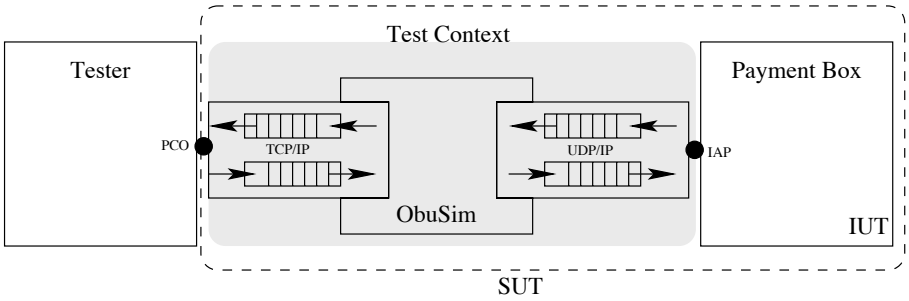

Figure 1.3 Test architecture

Test architecture implementation. The ObuSim is part of the automated test execution environment developed by Interpay B.V for testing of the PB. The major implementation activity for the test environment was extending this ObuSim with the interface by which TorX can control it. We also developed and parameterized a generic TorX Adapter component that endecodes the abstract specification actions on the message format of this ObuSim control interface.

SUT specification. A UDP/IP service can be modeled as queues, assuming that in our laboratory setting the service is FIFO and reliable. So, a first refinement of the specification is the addition of queues to our model [8]. As previously explained, we should add the behaviour of the ObuSim to the specification from which we derive the tests. The full test architecture is depicted in Figure 1.3. The tester communicates with PB via the test context (TCP/IP service, ObuSim,UDP/IP service). The PB together with this test context composes the SUT.

Testing the test architecture. Before we can start the "real" test execution, we need confidence in the correctness of both the specification and the test architecture implementation. To gain this confidence we use an iterative debugging process, in which we alternate between running TorX in manual mode and repairing inconsistencies in the specification (where changes made to abstract actions necessitate corresponding changes in the Adapter), and bug fixing the test architecture implementation. In this phase we encountered many problems.

3.4 TEST EXECUTION

Here we report our experiences during the test execution phase.

Introduction to campaigns. In the test environment debugging phase of our process we create multiple versions of specifications and Adapters, and run experiments using multiple configurations. When the test environment is ready and real testing starts, we perform many test executions with varying tool, specification, and possibly also implementation configurations. Previous case studies (e.g. [1]) have shown that it is a laborious task to keep track of the relation between execution results and the tool configuration (including specification and Adapter versions).

We set up a semi automatic bookkeeping system to manage the execution phase. In a *test campaign specification language* we specify a *test campaign* which contains all configuration details needed for automatic execution together with archiving information of the execution results. A test campaign tool supports the automatic execution of the specified campaign, i.e. the sequential execution of all experiments specified in the campaign. Related work has been reported in [10].

Campaign design. We designed a test campaign using a single (parameterized) Promela specification of the SUT. The configurations in the campaign vary over two parameters: 1) whether or not error stimuli are included, and 2) 78 different values for the random number seed parameter of TorX. So, the campaign describes 2 times 78 executions.

Campaign execution. Due to a minor tool problem (now fixed), and time constraints, of this test campaign only two test executions were performed, consisting of resp. 56167 and 27718 test steps.

Result analysis. During testing we found one conformance violation of the PB. At a certain moment the PB was silent (quiescence), while we expected output. This failure is still under analysis by Interpay B.V. Other conformance violations have not been found.

4 PRACTICAL ISSUES

When we executed the case study, we encountered many difficulties related to the gaps between theory, practice and tool support. We describe some problems in this section.

Model representation. The model of the SUT is decomposed into two main processes: the PB and the ObuSim. Inside both processes we made a decomposition for every transaction, i.e. every possible transaction is modeled by a separate process. This approach is most convenient since it makes the model finite, which is required for both the LOTOS and the PROMELA tools. Additionally, we benefit from the abstraction mechanism of parallelism. Unfortunately, during experimenting with many concurrent transactions, it turned out that for the relatively small model of about 50 processes the LOTOS primer did not perform fast enough to meet the timing constraints of the PB. For the rest of the experiment we used the PROMELA primer, which performed better.

Instantiation of parameters. When we test a system, concrete values have to be given for all data fields of all messages sent to the IUT. However, on specification level, we may abstract from values (free variable) in such input messages, for reasons of generality. Such an abstraction is often applied in writing formal specifications of systems, e.g. for model checking. Here, we have chosen to specify concrete values explicitly in the specification of the PB, to avoid having to instantiate with them elsewhere in the tester (e.g. in the ADAPTER).

Preprocessing specifications. Unfortunately, PROMELA and LOTOS do not support structuring concepts on meta level, e.g. to instantiate a number of processes or to include/exclude a partial behaviour description. As workaround we have used the M4 macro preprocessor [7]. It eases syntactical structuring and consistency keeping of the specification, and offers flexibility when the specification is used for test derivation.

Satisfying time-outs. The PbObu protocol contains time-out scenarios. When an expected message is not received within t_0 seconds, the PB will respond with a timeout message. A typical example of this behaviour is depicted in Figure 1.4.a. After the reception of the message `SendA` by the PB, the PB expects the message `SendB`. If this message does not arrive within t_0 seconds, the PB responds with the message `TimeOut`. The timeout value t_0 (less than 100 milliseconds) is very short compared to the speed with which TORX can deliver test primitives. Especially when we test for many parallel transactions, it can happen that

between the delivery of `SendA` and `SendB`, many other messages of other transactions are exchanged, so the chance of delivery within the time bound t_0 is very low. To solve this problem, we introduce an "artificial" event `SendAB` and refine it in the ObuSim. This refinement translates the new `SendAB` event to a transmission of `SendA` immediately followed by `SendB`. However, for various reasons, it can still happen that an error message will be sent after the `SendA` message. Because we can only observe after we have finished sending a stimulus, and the refinement always sends both `SendA` and `SendB`, the observation of this possible error message is delayed, which we have to model explicitly. We do that by adding a queue to the model (Figure 1.4.b).

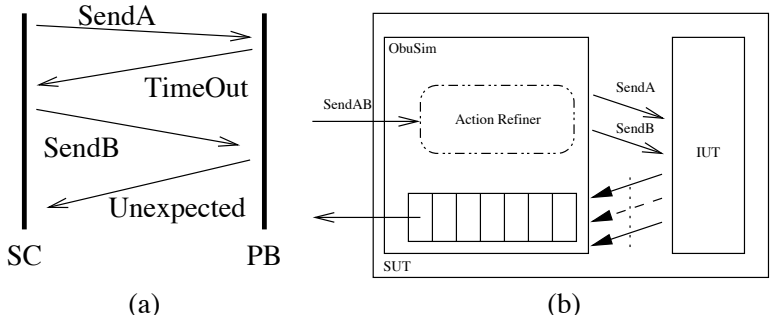

Figure 1.4 Timing and action refinement

Quiescence and time-outs. The observation of absence of output (quiescence) is implemented by starting a timer when trying to observe any output. If the timer expires after t_q seconds we expect that no output will occur anymore and conclude that we observed quiescence (δ). However, this assumption is too strong for testing the PB. If we choose t_q longer than the timeout t_0 of the PB, we will never make progress in the protocol, i.e. never exercise all behaviour. We prefer to choose t_q shorter than t_0. As a result we might conclude that the PB fails for conformance since after the observation of δ we conclude that we cannot detect any possible output anymore. However, the `TimeOut` error message may occur after that δ observation. As a pragmatic solution we introduce an additional action `Tick`. When the IUT is in a state with a time-out scenario, i.e. expects an input, then we generate a `Tick` output action as long as no other output occurs. The `Tick` action is explicitly modeled in the specification.

Guarded inputs. Initially we specified parts of the behaviour by a state-oriented machine in PROMELA, i.e. transitions are enabled based

on the value of a (global) state variable. However, PROMELA internally translates a predicate into an internal event that blocks based on the evaluation of the predicate, instead of being a blocking condition of an observable action. As a result, the set of actions predicted by the specification is incorrect (this goes wrong with the quiescence predictions), leading to invalid tests. Based on our notion of the semantics of PROMELA and the **ioco** theory, we avoid the use of predicates as internal events. We specified the SUT in a non state-oriented style. Unfortunately this extends the state exploration space.

Probabilistic choice. When running TORX in automatic mode, we noticed that its random input selection strategy (that gives all input actions the same chance of being chosen) did not give us the tests we were interested in. The input actions in our specification fall apart into two categories: one for normal behaviour, and one for error behaviour. In our tests, we do want the main focus on normal behaviour, without completely ignoring error behaviour. The random selection strategy of TORX pays too much attention to error behaviour. To solve this problem, we added the PARTITIONER tool component to TORX, and configured it such that error related actions have a lower chance of being selected. The open architecture of TORX (and the textual interfaces in particular) allowed integration of this new tool component.

Results review. The first execution consisted of 56167 test steps. Of the 17099 transactions which were started, 2200 were successfully completed, 10676 did not meet the timing constraints of the PB, and 4211 did not succeed due to permanent or transitional (21) problems with the smart cards, e.g. insufficient balance. At the end of the execution 12 transactions were still in progress. No error was detected in the PB. Only 61 `Tick` actions occurred, and only once quiescence was observed.

For the second execution, consisting of 27718 test steps, these numbers are respectively 8429 started, 1195 completed, 5237 time constrained, 1992 smart card problems, and 5 still in progress; 24 `Tick` actions, and one quiescence observation. No error was detected in the PB.

The large number of transactions that timed out can be explained by the large number of transactions occurring in parallel, combined with the problems that we had in constructing a specification from which the test events can be derived fast enough.

The low number of `Tick` actions can be explained by the small period of time in which the PB is in a state in which it is waiting for input while all output produced by the PB already has been consumed. The even lower number of quiescent observations is explained in the same way.

5 RESULTS AND OBSERVATIONS

We described automated test derivation and test execution for conformance testing of the PB. We described stepwise the actions that are involved in setting up an automatic on-the-fly test environment and the difficulties and peculiarities that we encountered. We evaluate these:

1. A formal specification for testing differs from a formal specification for validation. This is based on two observations. Firstly, a specification for validation is a model of the IUT. For test derivation we need a model of the SUT. Secondly, a specification for validation can abstract from details which are not of interest for showing a particular property of the system. A specification for conformance testing should describe all possible behaviour at the abstraction level of stimulation and observation.

2. One of the expected benefits of automated derivation, its flexibility, was demonstrated during the engineering phase of the test environment. Since it was hard to identify the SUT in advance , the SUT specification was developed iteratively during the test phase of the test environment and the engineering of the ObuSim. Adaptations of behaviour, needed to resolve misinterpretation of the informal specification, or to derive specific experiments for validation of the test environment, could easily be solved on specification level, and the proof testing could proceed immediately, since we did not have to rewrite test cases.

3. The flexibility of automated derivation was also shown during testing. By small adaptations of the specification we could easily scale up the experiment (increase the number of concurrent transactions) and direct the tests to a particular test goal. An example of the latter is the disabling of the error generation by the environment, resulting in the increased probability of successful electronic payments transactions. Another issue in the flexibility was shown at the start of the case study. Since we have a formal specification, debugging the specification was easy thanks to the tool support of Spin and Cæsar. During the model checking phase, we encountered a functional error in the protocol, which would lead to erroneous money balances. This error has been fixed and new specifications of the HTS have been released.

4. The development of the specification is an iterative process. During the implementation of the Adapter more detail of the representation of abstract actions gets known, and our understanding of the SUT increased in the test environment debugging phase.

5. In this case study timing constraints, i.e. handling of time-outs, are dealt with in an ad hoc way. The **ioco** test theory does not offer any theory for handling time-out; neither does TorX. In this application

we encountered problems related with the observation of quiescence and specifying time-outs. Since many reactive systems, e.g. protocols, have time-out behaviour, more fundamental theory is needed for testing this real-time class of systems.

6. The assumption that we can synthesize every stimulus and analyze every observation is strong. Because of the encryption involved we delegated the synthesis and analysis to the ObuSim. To resolve timing constraints we used action abstraction and action refinement. More theoretical background on the synthesis of test primitives and action abstraction and refinement in testing is needed to implement generic Adapters for automated test environments.

7. The on-the-fly test approach is challenged when testing systems with real time requirements where the computation of test primitives has to meet certain time bounds to meet an IUT's timing constraints. Although we used an ad hoc solution to cope with the real-time requirements of the PB (stimulate in time), we were not able to generate test primitives fast enough using a Cæsar Primer. A reason could be that the rendez-vous synchronization, where more than 20 processes were involved, slowed down the computation of test primitives. The Promela Primer performed better. Fast computation of test primitives requires efficient handling of the state graph by the Explorer.

8. The automated test derivation from system specifications with a high degree of parallelism is fast and reliable, compared to traditional manual test derivation, which is complex due to the bookkeeping of all interleaving scenarios. By parameterizing the specification we could easily scale up the experiment with respect to the number of concurrent transactions. TorX can handle models of concurrent systems well.

9. The engineering of the automated test execution environment, i.e. the implementation of the Adapter is a laborious task. The hard part was to synthesize the concrete test primitives, i.e. dealing with the smart cards. This conclusion is supported by the conclusions of the traditional test of the PB by Interpay B.V. Also there the whole test execution environment was automated and the implementation of the automated environment was very laborious. Further decomposition into domain specific components of the Adapter can decrease the labour of the development of the application dependent Adapter.

10. The TorX architecture is very flexible. Due to the standardized textual interface between Driver and Adapter the implementation effort was minimal with respect to this interface. Furthermore, it was easy to cope with multi-platform constraints, i.e. TorX running on a

Unix workstation and the ADAPTER on a Windows NT computer. The other TORX components could be reused easily.

11. Due to automation we can execute many tests fast. Since we may use different specifications, TORX tool instantiations (in this case not fully utilized), configurations etc., we will obtain many test executions and test results. Adding the dimension of time, e.g. different releases of the system, specifications etc., even increases the amount of test data. To cope with this complexity we started structuring the test executions and their parameters into test campaigns for which we developed prototype tool support. Initial experiments using campaigns and its tool support demonstrate both the feasibility of such test execution management, and the need for more development on practical and theoretical level.

12. The test campaigns currently only specify (and capture) the elements of the testing process that can be formalized and controlled automatically. Of course, this is not sufficient. Sometimes it is needed to extend the formal experiment description with plain text descriptions of critical test environment parameters that can only be influenced (set up before testing starts) by a human test operator.

13. The results obtained from the execution of a test campaign are large in size and comprehensive to interpret. We need advanced tool support for in-depth analysis of the results, to cope with the complexity.

6 CONCLUDING REMARKS

Summarizing all partial conclusions, we conclude that automated formal test derivation and test execution is beneficial because of the high number of tests that we can execute, and the reliability of these tests (8). Furthermore, automation is feasible due to its flexibility. Adaptations and parameterization of specifications, and test environment configurations are easy (2,3,10). We see the testing of Payment Box as a step ahead in testing realistic industrial applications. However, we need more techniques, theory and tools to support automated testing of real industrial applications. On theory level we need more support for real time behaviour (5,7). On technical and tool level we need more support for system specification (1), result analysis (11,12) and ADAPTER implementation (4,6,9).

Acknowledgments

The authors thank Cornel van Mastrigt and Rommert Jorritsma from Interpay B.V. for their help and support. The anonymous reviewers and Jan Tretmans are acknowledged for their comments.

References

[1] 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*. Kluwer Academic Publishers, 1999.

[2] L.M.G Feijs, N. Goga, and S. Mauw. Probabilities in the TorX test derivation algorithm. In Claude Jard Susanne Graf and Yair Lahav, editors, *SAM2000 - 2nd Workshop on SDL and MSC*, pages 173–188, Col de Porte, Grenoble, 6 2000. VERIMAG, IRISA, SDL Forum Society.

[3] 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.

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

[5] 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.

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

[7] René Seindal. *GNU m4, version 1.4*. Free Software Foundation, 59 Temple Place – Suite 330, Boston, MA 0211, USA, 1.4 edition, November 1994. Available from URL: http://www.gnu.org.

[8] R. Terpstra, L. Ferreira Pires, L. Heerink, and J. Tretmans. Testing theory in practice: A simple experiment. In T. Kapus and Z. Brezočnik, editors, *COST 247 Int. Workshop on Applied Formal Methods in System Design*, pages 168–183, Maribor, Slovenia, 1996. University of Maribor.

[9] 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.

[10] T. Vassiliou-Gioles, I. Schieferdecker, M. Born, M. Winkler, and M. Li. Configuration and execution support for distributed tests. In K. Tarnay G. Csopaki, S. Dibuz, editor, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 61–76. Kluwer Academic Publishers, 1999.