ON-THE-FLY FORMAL TESTING OF A SMART CARD APPLET

Arjen van Weelden, Martijn Oostdijk, Lars Frantzen, Pieter Koopman, Jan Tretmans

{arjenw, martijno, lf, pieter, tretmans}@cs.ru.nl
Institute for Computing and Information Sciences
Radboud University Nijmegen - The Netherlands

Abstract: This paper presents a case study on the use of formal methods in specification-based, black-box testing of a smart card applet. The system under test is a simple electronic purse application running on a Java Card platform. The specification of the applet is given as a Statechart model, and transformed into a functional form to serve as the input for the on-the-fly test generation, -execution, and -analysis tool GAST. We show that automated, formal, specification-based testing of smart card applets is of high value, and that errors can be detected using this model-based testing.

Key words: model-based testing; smart cards; Java Card; automatic test generation; executable specification.

1. INTRODUCTION

Smart devices are often used in critical application domains, such as electronic banking and identity determination. This implies that their quality, such as their safety, security, and interoperability, is very important. Such devices commonly implement a Java Card virtual machine, which is able to execute Java Card applets. Each application is then implemented as a separate applet.

One way to increase the quality of applets is the use of formal methods. Such applets are sufficiently small to make a complete formal treatment with current day formal technology feasible. Systematic testing is another method, predominantly used to check the quality of smart devices in an experimental way.

In this paper we combine testing and formal methods: we test a Java Card applet, in a black-box setting, based on a formal specification of its required behavior. Compared with formal verification, testing has the advantage that it examines the real, complete system consisting of applet, platform and hardware together, whereas formal verification is usually restricted to a model of the applet only. Compared with traditional, manual testing, formal testing has as first advantage that the formality reduces the ambiguities and misinterpretations in the specification so that it is clearer what should be tested. Secondly, formal specifications allow to completely automate the testing process: test cases are algorithmically generated from the formal specification, and test results can automatically be analyzed. This makes it possible to generate and execute large quantities of large tests in a short time. It is mainly this second advantage that we will pursue in this paper.

Our investigation of formal testing is conducted using a case study. The applet that we test is a simple electronic purse application, implemented as a Java Card applet, with a limited set of methods like asking the value on the card, debiting, crediting, etc. The formal specification of the electronic purse applet is given as a Statechart[1]. The case and its formal specification are described in Section 2.

Automatic test generation, test execution, and test result analysis are performed in an *on-the-fly* fashion, using the test tool GAST[2]. The test tool GAST is described in Section 3. Section 4 describes how the Statechart specification is transformed into Clean[3], a functional programming language used as the input language for GAST. How GAST, the applet under test, and the platform are connected is described in the test architecture, which is given in Section 5.

We constructed one (assumed to be) correct applet implementation. From this implementation we derived 22 mutants by inserting subtle bugs to see whether such bugs would be detected by our automated, formal testing method. A summary of the performed tests is given in Section 6. Finally, Section 7 and 8 discuss related work, conclusions, and possible future extensions.

2. CASE STUDY

We demonstrate our testing methodology by applying it to a simple electronic purse application as a case study. The basic events that the electronic purse can receive are:

- set an initial value `n` via `setValue(n)`
- query the actual value via `getValue()`
- pay an amount of `n` via `debit(n)`

- authenticate with a `pin` (personal identification number) via `authenticate(pin)` before charging the card
- charge the card with an amount of `n` via `credit(n)`
- reset the card using a `puk` (personal unlocking key) via `reset(puk)`

All these events are input events for the card, because they are sent from the Card Accepting Device (CAD, also called *terminal*) to the card. To every input event, the card answers with a corresponding output event:

- acknowledge an operation via `ackOK` or `ack(n)`
- report an error via `error(n)`

Figure 1 shows the specification of the purse, modeled as a Statechart.

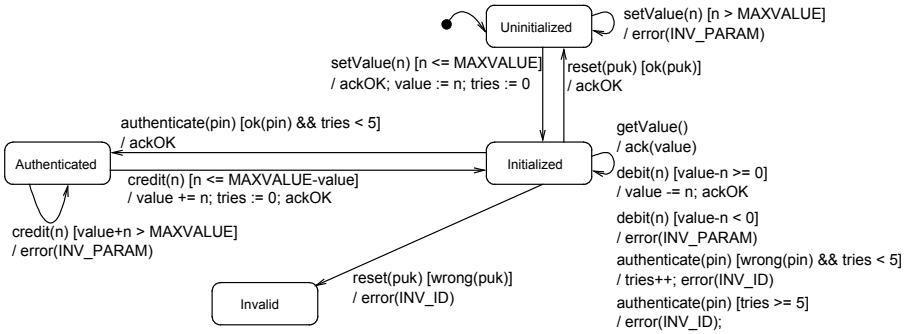

Figure 1. Statechart model of the purse applet.

The transition labels between two states s_1 and s_2 are of the form:

$$s_1 \xrightarrow{i[g]/act} s_2$$

with i being an input event, g being a guard, and act representing a sequence of actions. We exemplify the semantics with this transition:

$$Autenticated \xrightarrow{credit(n)[n \leq MAXVALUE - value]/\, value += n; tries := 0; ackOK} Initialized$$

The input event (i) is `credit(n)`. The argument `n` represents the amount of money to be added to the card. The applet uses signed 16-bit integer `shorts` and it gives an `error(INV_PARAM)` on negative values. We abstract from that in the Statechart to keep it concise. The actual value of the card is saved in the variable `value`. A transition can only fire when the corresponding guard g holds. In this example, one can only increase the value of the card by `n`, when `n` does not exceed the `MAXVALUE-value`. If the transition is taken, the actions *act* are performed. In this case, the variable

`value` is incremented by `n`, the `tries` variable is reset to zero, and the acknowledgment `ackOK` is sent to the terminal.

Intuitively, the purse works as follows. At first, the card is in the `Uninitialized` state. It is initialized by the credit institution, which issues the card to the customer by putting a certain amount `n` of money on it via the `setValue(n)` event.

In the `Initialized` state the customer can query the actual value via the `getValue()` event, or pay with the card via the `debit(n)` event. To increase the value, one must first authenticate at a terminal with a card-specific `pin`, leading to the `Authenticated` state. Being in that state, one can add money via the `credit(n)` event, leading back to the `Initialized` state. The card checks that its value does not exceed the `MAXVALUE`.

Furthermore, there is a maximum of five tries to enter the `pin`. After the fifth wrong attempt, one can no longer credit the card. If the credit institution enters the reset code (called `puk`) correctly, the card goes back to the `Uninitialized` state and can be re-initialized via the `setValue(n)` event. If the `puk` is entered wrongly, the card goes to the `Invalid` state and cannot be used anymore.

Two kinds of erroneous events can be sent to the card. Firstly, a syntactically correct input event that is not specified for the actual state may occur, e.g., a `credit(n)` when the card is in the `Initialized` state. Such an unspecified input event is called an *inopportune event*, and the response of the applet should be an error message `error(INV_CMD)`, whereas the applet remains in its actual state. Secondly, a syntactically incorrect event may occur, e.g., a command-APDU with a non-existing event-code. This is also implicitly assumed to lead to an error message, while the card stays in its current state.

3. THE TEST TOOL GAST IN A NUTSHELL

The test tool GAST is designed to be open and extendable. For this reason it is implemented as a library rather than a standalone tool. The functional programming language Clean is chosen as host language due to its expressiveness.

GAST can handle two kinds of properties. It can test properties stated in logic about (combinations of) functions. GAST can also test the behavior of reactive systems based on Extended (Finite) State Machines, E(F)SM. Here we will only discuss the ability to test reactive systems.

An ESM as used by GAST comes quite close to the Statechart of Figure 1. It consists of states with labelled transitions between them. A transition is of the form $s \xrightarrow{i/o} t$, where s, t are states, i is an input which triggers the

transition, and o is a, possibly empty, list of outputs. The domains of the inputs, outputs, and states can be given by arbitrarily complex recursive Algebraic Data Types (ADT). This constitutes the main difference with traditional testing with FSM's where the testing algorithms can only handle finite domains and deterministic systems[4].

A transition $s \xrightarrow{i/o} t$ is represented by the tuple (s, i, t, o). A relation based specification δ_r is a set of these tuples: $\delta_r \subseteq S \times I \times S \times O^*$. The transition function δ_f is defined by $\delta_f(s,i) = \{ (t, o) \mid (s, i, t, o) \in \delta_r \}$. Hence, $s \xrightarrow{i/o} t$ is equivalent to $(t, o) \in \delta_f(s, i)$. A specification is *partial* if for some state s and input i we have $\delta_f(s, i) = \varnothing$, otherwise it is *total*. A specification is *deterministic* if for all states and inputs the size of the set of targets contains at most one element: $\#\,\delta_f(s, i) \leq 1$. A *trace* σ is a sequence of inputs and associated outputs from a given state. Traces are defined inductively: the empty trace connects a state to itself: $s \xrightarrow{\varepsilon} s$. We can combine a trace $s \xrightarrow{\sigma} t$ and a transition $t \xrightarrow{i/o} u$ form the target state t, to trace $s \xrightarrow{\sigma;i/o} t$. We define $s \xrightarrow{i/o} \equiv \exists t.s \xrightarrow{i/o} t$ and $s \xrightarrow{\sigma} \equiv \exists t.s \xrightarrow{\sigma} t$. All traces from state s are: $traces(s) \equiv \{\sigma \mid s \xrightarrow{\sigma}\}$. The inputs allowed in state s are given by $init(s) = \{i \mid \exists o.s \xrightarrow{i/o}\}$. The states after trace σ in state s are given by s after $\sigma \equiv \{t \mid s \xrightarrow{\sigma} t\}$. We overload *traces*, *init*, and after for sets of states instead of a single state by taking the union of the individual results. When the transition function, δ_f, to be used is not clear from the context, we will add it as subscript.

The basic assumption for our formal testing is that the Implementation Under Test, iut, is also a state machine. Since we do black box testing, the state of the iut is invisible. The iut is assumed to be *total*: any input can be applied in any state. Conformance of the iut to the specification spec is defined as (s_0 is the initial state of spec, and t_0 of iut):

$$\text{iut } conf \text{ spec} \equiv \forall \sigma \in traces_{\text{spec}}(s_0), \forall i \in init(s_0 \text{ after}_{\text{spec}}\ \sigma), \forall o \in O^* \bullet$$
$$(t_0 \text{ after}_{\text{iut}}\ \sigma) \xrightarrow{i/o} \Rightarrow (s_0 \text{ after}_{\text{spec}}\ \sigma) \xrightarrow{i/o}$$

Intuitively: if the specification allows input i after trace σ, the observed output of the iut should be allowed by the specification. If spec does not specify a transition for the current state and input, anything is allowed. This notion of conformance is very similar to the ioco relation[5] for Labeled Transition Systems (LTS). In an LTS each input and output is modeled by a separate transition. In our approach an input and all induced outputs up to *quiescence* are modeled by a single transition. Quiescence characterizes a state of the iut that will not produce any output before a new input is provided, i.e. a quiescent system waits for input and cannot do anything else.

In order to test conformance, a collection of input sequences is needed. At the beginning of each input sequence GAST resets the iut and the spec to their initial state. By applying the inputs of a sequence one by one, GAST investigates if it can be transformed to a trace of spec. The previous inputs and observed responses are remembered in trace σ. If $\delta_{spec}(s, i) \neq \varnothing$ for the current input i and some state s reachable from the initial state, S_0, by trace σ (i.e. $s_0 \xrightarrow{\sigma} s$), the input is applied to the iut, and the observed output is checked to be allowed by spec.

GAST has several algorithms for input generation, e.g.:

- Systematic generation of sequences based on the input type.
- Sequences that cover all transitions in a *finite* state machine.
- Pseudo random walk through the transitions of a specification.
- User defined sequences.

In this paper we will only use the third algorithm to generate input sequences. Testing is *on-the-fly*, which means that input generation, execution, and result analysis are performed in lockstep, so that only the inputs actually needed will be generated. The lazy evaluation of Clean used for the implementation of GAST makes this easy.

Within the test tool GAST, the mathematical state transition function, δ_f, specifying the desired behavior is represented by a function in the functional programming language Clean. Functional languages allow very concise representations of specifying functions and have well understood semantics. Using an existing language as notation for the specification prevents the need to design, implement and learn a new language. The rich data types and available libraries enable compact and elegant specifications. The advanced type system of functional languages enforces consistency constraints on the specification, and hence prevents inconsistencies in the specification.

Since the specification is a function in a functional programming language, it can be executed. This is convenient when one wants to validate the specification by observation of its behavior.

Any Clean type can be used to model the state, the input and the output of the function specifying δ_f, including user-defined data types. GAST uses generic programming techniques for generating, comparing, and printing of these types. This implies that default implementation of these operations can be derived without any effort for the test engineer. Whenever desired, these operations can be tailored using the full power of the functional programming language.

4. THE PURSE SPECIFICATION FOR GAST

The specification given in the Statechart is transformed to the functional language Clean in order to let GAST execute and manipulate it. This section gives some details about the representation in Clean of the electronic purse from Figure 1. Due to space limitations we will only show snapshots of the (executable) specification. A parameterized enumeration type represents the state of the purse

```
:: PurseState = Uninitialized | Initialized Short Short
              | Authenticated Short | Invalid
```

We use one constructor for each state from the Statechart in Figure 1. The arguments of the constructor `Initialized` represent the tries counter and the value. The type `Short` represents signed 16-bit integers. This implies that there are actually $2^{16} \times 2^{16} = 2^{32}$ different `initialized` states, of which some are not reachable. There are similar types for input and output.

A transition function `purse`, similar to δ_f, in Section 3 models the transitions. The only difference with the mathematical specification is that the result is a list of tuples instead of a set of tuples. Some function alternatives specifying characteristic transitions are:

```
purse :: PurseState PurseInput -> [(PurseState, [PurseOutput])]
purse Uninitialized (SetValue n)
    = if (n >= 0 && n <= MAXVALUE)
        then [(Initialized 0 n, [AckOK])]
        else [(Uninitialized, [Error INV_PARAM])]
purse (Initialized tries value) Reset
    = [(Uninitialized, [AckOK])]
purse (Initialized tries value) GetValue
    = [(Initialized tries value, [Ack value])]
...
purse state any = [(state, [Error INV_CMD])]
```

The first alternative models both transitions for the input `SetValue n` from the state `Uninitialized`. The second and third alternative show two transitions form the state `Initialized`. The last alternative captures the informal requirement that inopportune events should cause no state transition and an error message as output. Since `state` and input `any` are variables, this alternative covers any combination not listed above. Since exactly one transition is defined for each combination of state and input, the specification is total and deterministic.

This specification is an ordinary definition in the functional programming language Clean. It is checked by the compiler before it is used by GAST. This guarantees well-defined identifiers and type correctness.

5. TESTING JAVA CARDS WITH GAST

The tests, which will be described in Section 6, have been executed using the test architecture of Figure 2.

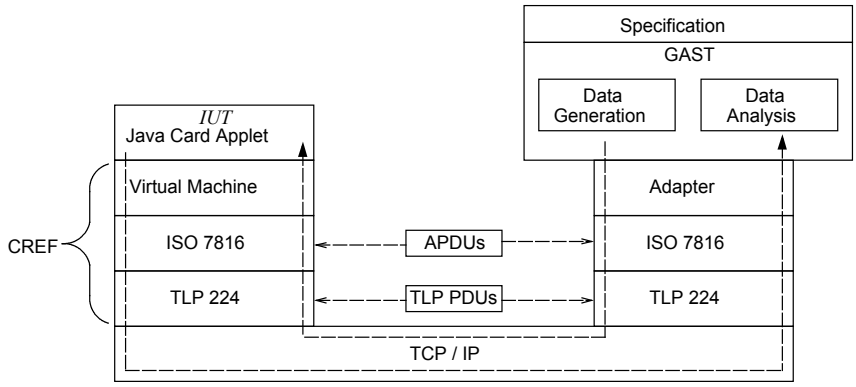

Figure 2. The general testing framework.

The IUT is the Java Card applet implementing our simple electronic purse. To make testing easier and more flexible, we used a simulation platform to execute the applet. The simulation environment is the *C-language Java Card Runtime Environment* (CREF), which comes with the *Java Card Development Kit*. CREF simulates a Java Card technology-compliant smart card in a card reader. It further consists of a Java Card Virtual Machine, and communication protocol entities to allow communication between the applet and the outside world.

To communicate with the applet under test, GAST was enhanced to be able to deal with these typical smart card communication protocols *ISO-7816-4* and *TLP-224* over TCP/IP. On top of these protocol entities an *adapter* (glue code), was implemented. The adapter transforms the high level inputs, generated by GAST, and represented as Clean data values, into the low-level APDUs, coded as appropriate byte codes, and then sent according to the ISO-7816-4 protocol. Vice versa, the adapter decodes the APDUs received from the applet under test to Clean data values, which are then analyzed and checked by GAST.

For data generation and analysis GAST uses the Clean EFSM specification, which was developed in Section 4. Except for the access to TCP/IP, the right-hand side of Figure 2 was entirely implemented in Clean.

The use of a simulation platform for testing is not a restriction with respect to testing of real smart cards. Since only standardized protocols are used, GAST cannot see the difference between testing on a simulator, and testing a real smart card. The test architecture could easily be adapted to test

real cards by swapping CREF with a real card and its reader. The use of a simulation platform does facilitate easy switching between different applets, and saving and restoring applet state.

6. RESULTS

The Statechart in Figure 1 and its implementation as an applet were developed in an incremental way. GAST appears to spot differences between the specified and actual behavior very rapidly. Once the specification and implementation were finished, the testing power of the GAST system was determined in a systematic way using *mutants*. Starting from the ideal (assumed to be correct) applet we injected typical programming errors into the applet, and analyzed how long it took GAST to find errors by generating, executing, and analyzing tests. The mutants are obtained by subtle changes like omitting checks or updates to the state of the applet. The test results for the 22 mutants used are listed in Table 1.

Table 1. Overview of test results.

nr.	paths	trans-itions	time (s)	inputs until error	comments
1	1	25	0.49	25	6 tries allowed in this mutant
2	2	66	0.09	31	incorrect overflow during `credit`
3	1	9	0.47	9	negative balance allowed in mutant
4	5	247	0.71	41	tries not reset after `authenticate`
5	8	406	0.38	51	tries not reset after `reset`
6	1	1	0.05	1	`credit` allowed without `authenticate`
7	1	1	0.52	1	`setvalue(0)` not allowed
8	1	4	0.06	4	`credit` with negative amount allowed
9	1	2	0.50	2	`debit` with negative amount allowed
10	11	542	0.48	23	no check for locked flag
11	7	327	0.80	26	not locked after 5 attempts
12	1	13	0.06	13	stays authenticated
13	21	1020	1.28	21	not locked after `reset`
14	1	16	0.09	16	`MAXVALUE` too low
15	1	24	0.07	24	`authenticate` does not authenticate
16	1	33	0.52	33	`reset` does not make it uninitialized
17	94	4757	3.82	29	`tries ≤ 5` instead of `tries < 5`
18	4	207	0.26	23	fresh card has nonzero balance
19	1	6	0.30	6	`setValue` allowed in initial state
20	3	145	0.18	44	`setValue` does not initialized/unlock
21	1	4	0.50	4	`MAXVALUE` too high
22	5	206	0.67	2	`MAXVALUE` balance not allowed
	7.8	366.4	0.56	19.5	averages
	100	5081	4.20	n/a	original applet, no counterexample

For instance, mutant 17 differs from the ideal applet by testing whether the number of remaining authentication tries is *less than or equal to* five rather than *less than* five before setting a flag indicating that the applet should no longer accept authentication attempts. This mutant was found after executing 94 paths, within 3.82 seconds, containing 4757 transitions in total. This mutant showed an invalid output after an input sequence of length 29 in path 94. To identify the error, the trace of inputs and associated responses are written to a file.

GAST was able to identify the 22 incorrect implementations without any help, using a minimum path length of 50 transitions and a maximum of 100 paths. It took an average of 0.56 seconds to generate and execute, on average, 366 transitions on a 1.4GHz Windows computer. Identifying incorrect behavior for all 22 mutants cost only 12 seconds in total. This shows that GAST is an efficient and effective test tool.

7. RELATED WORK

Two approaches are closely related to ours due to the fact that both rely on tools that implement variants of the ioco testing relation[5]. Du Bousquet and Martin[6] use UML specifications, which are translated into Labeled Transition Systems to serve as input for the TGV tool[7]. Instead of an on-the-fly execution, TGV uses additional test purposes to generate test cases. The authors created a tool to automate the generation of test purposes based on common testing strategies. The generated test cases are finally translated into Java code, which communicates with the applet and executes the test. TGV does not treat data symbolically, which can easily lead to a state space explosion when dealing with large data domains. Because we generate test cases on-the-fly based on the (symbolic) EFSM, this problem does not occur.

To support symbolic treatment of data, Clarke et al.[8] use Input/Output Symbolic Transition Systems. The basic approach is similar to TGV, hence also here test purposes are needed. The test automation is done via a translation to C++ code, which is linked with the implementation. This restricts the IUT to be a C++ class with a compatible interface.

Rather than *testing* properties of the IUT, its implementation (i.e., the Java Card applet) can also be formally *verified*. Testing and verification are complementary techniques to check the correctness of systems, as explained in Section 1. A common technique used for verifying Java Card applets is to prove their correctness with respect to a specification in the Java Modeling Language (JML). State-based specifications similar to the one in Figure 1 can uniformly be translated to JML specifications, as shown by Hubbers, Oostdijk, and Poll[9]. The resulting annotated Java Card applet can then be

verified using one of the many JML tools[10], for instance, the ESCJava2 static analyzer[11]. Most Java Card applets are small enough to even attempt a formal correctness-proof using the Loop tool, as demonstrated by Jacobs, Oostdijk, and Warnier[12].

8. CONCLUSION AND FUTURE WORK

We have presented an approach to automate the testing of Java Card applets using the test tool GAST. The test case derivation is based on a Statechart specification of the applet under test. The specification can directly be translated into a corresponding GAST specification. Tests were completely automatically derived, executed, and analyzed. Discrepancies between the formal specification and its Java Card implementation were successfully detected, which shows the feasibility of this approach.

The direct translation from the Statechart model to the GAST specification, and the on-the-fly execution of the test cases enable the developer to start with automatic testing of the applet in the early stages of development. The co-development of the formal model and the implementation, and the facility to do automatic tests, has shown to be very useful. Both the code and the specification have evolved simultaneously, vastly improving the quality of the applet, and leading to a complete and reliable specification. Such a specification delivers insight on how to specify similar cases, and can serve as a pattern for these.

The tested mutants, representing typical programming errors, have increased our confidence in the error detecting power of the GAST algorithm. We are planning to compare this with other test tools, e.g., the ioco-based tool TorX[13], to test more complex applets, testing applets on real cards, and testing advanced aspects like the integration, interference, and feature interaction between different applets on one card.

Finally, we will compare the testing approach with the formal verification approach, e.g., using JML, to see how far we can get in unifying verification and testing techniques into one common framework, and to investigate the precise shape of their complementarities.

9. REFERENCES

1 UML resource page. `http://www.uml.org`.

2 P. Koopman and R. Plasmeijer. Testing reactive systems with GAST. In S. Gilmore, editor, *Trends in Functional Programming 4*, 111-129 (2004)

3 R. Plasmeijer and M. van Eekelen. *The Concurrent Clean Language Report*, version 2.0. `http://www.cs.kun.nl/~clean`.

4 D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proc. IEEE*, 84(8):1090--1126 (1996)
5 J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software-Concepts and Tools*, 17(3):103-120 (1996)
6 L. du Bousquet and H. Martin. Automatic test generation for Java-Card applets. In *4th Workshop on Tools for System Design and Verification*, (2000)
7 C. Jard and T. Jéron. TGV: theory, principles and algorithms. In *IDPT '02*, Pasadena, California, USA, Society for Design and Process Science (2002)
8 D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *Proceedings of the International Conference on Research in Smart Cards*, volume 2140 of *LNCS*, 58-70, Cannes, France (2001)
9 E. Hubbers, M. Oostdijk, and E. Poll. From finite state machines to provably correct java card applets. In D. Gritzalis, S. De Capitani di Vimercati, P. Samarati, and S.K. Katsikas, editors, *Proceedings of the 18th IFIP Information Security Conference*, Kluwer Academic Publishers, 465-470 (2003)
10 L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In Th. Arts and W. Fokkink, editors, *FMICS '03*, volume 80 of *ENTCS*, pages 73-89 (2003)
11 D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML: progress and issues in building and using ESC/Java2. Submitted for publication (2004)
12 B. Jacobs, M. Oostdijk, and M. Warnier. Source code verification of a secure payment applet. *JLAP*, 58:107--120 (2004)
13 J. Tretmans and E. Brinksma. TorX: Automated model based testing. In A. Hartman and K. Dussa-Zieger, editors, *First European Conference on Model-Driven Software Engineering*. Imbuss, Möhrendorf, Germany (2003)