On-the-Fly Formal Testing of a Smart Card Applet

A. van Weelden, L. Frantzen, M. Oostdijk, P. Koopman, J. Tretmans

Nijmegen Institute for Computing and Information Sciences/

NIII-R0428 June 2004

Nijmegen Institute for Computing and Information Sciences
Faculty of Science
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

On-the-Fly Formal Testing of a Smart Card Applet

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

Nijmegen Institute for Computing and Information Sciences (NIII)
Radboud University Nijmegen – The Netherlands
{arjenw,lf,martijno,pieter,tretmans}@cs.kun.nl

Abstract. Smart cards are used in critical application areas. This means that their software should function correctly. Formal methods are indispensable in obtaining high quality systems. 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 test generation, -execution, and -analysis tool Gast. Several test runs were conducted, completely automatically, and altogether consisting of several millions of test events. The tests were applied on an (assumed to be) correct applet implementation, as well as on some error-seeded implementations. They showed that automated, formal, specification-based testing of smart card applets is feasible, and that errors can be detected.

1 Introduction

Smart cards Smart devices play an increasingly important role in many applications in electronic banking, telecommunication, identity determination, etc. These kind of applications put high requirements on the quality of such devices, in particular on properties like safety, security, and interoperability. This even more applies to open smart devices where multiple applications can dynamically be put on a single device, which may lead to many forms of intended and unintended interaction between those applications. These multi-application cards commonly contain a Java Card virtual machine, which is able to execute Java Card applets [Che00]. Each application is then implemented as a separate applet. In this paper we will consider the functional correctness of such a Java Card applet: a simple electronic purse application.

Formal methods Formal methods refer to the use of mathematical and logical techniques to specify, model, and reason about systems. This allows to give formal proofs of properties, e.g., safety or security properties, with mathematical rigor, thus increasing the confidence in the correct functioning of systems. Formal methods are expected to play an increasingly important role in guaranteeing the quality of smart devices. Smart devices, on the one hand, are used in very critical and secure environments so that their correctness is of utmost importance. On the other hand, smart cards and the applications running on them are sufficiently small to make a complete formal treatment with current day formal technology feasible.

The behavior of our simple purse application is modeled with a Statechart [UML], which, in turn, is expressed in the formal, functional language CLEAN [PvE]. This CLEAN expression then serves as the formal specification of the simple purse application.

Testing Testing is an important, in practice even the dominating technique to check the correctness of systems. Testing consists of performing, in a controlled way, experiments on a system under test in order to see whether the system has the desired properties. Traditionally, testing is performed using informal methods, i.e., tests are manually developed based on some informal, natural language description of the system specification, and also test results are manually analyzed and judged. Such a testing process is laborious and error-prone, and moreover the natural language specifications often lead to ambiguities and misinterpretations about the system's required properties. And, if the requirements are not clear, it is, of course, difficult to test whether the system does what it should do.

Formal testing In formal testing the starting point is a formal specification, which prescribes the required properties of the system under test. The system under test itself is seen as a black box, without internal detail, and communicating with its environment via its interfaces. Testing is used to check whether the system has the formally specified properties. To achieve this, test cases are algorithmically generated from the formal specification. A test case specifies the experiment to be performed to check a property, or part of a property. Since the system under test is seen as a black box, the test must be expressed in terms of stimuli (inputs) and responses (outputs) on the system's interfaces. During the execution of the specified experiment, preferably using some automated test environment, the test results are analyzed, again algorithmically, with respect to the formal specification.

Formal testing alleviates two problems of traditional testing. First, the formality of the specification reduces the ambiguities and misinterpretations about the system's required properties. Secondly, formal specifications allow algorithmic generation of test cases and analysis of results, thus making it possible to automate the whole testing process. That is what we will pursue in this paper.

There are different approaches to formal testing, mostly depending on the formalism in which the specification is expressed. Examples are testing with finite state machine specifications [LY96], testing where the program under test is seen as a function [KATP03], testing based on transition system models [Tre99], testing using abstract data type specifications [Gau95], model based testing using abstract state machines [BGN$^+$04], and others. For most of these approaches test tools have been developed to support and automate the formal testing process [BFS04].

Testing and verification Formal verification and testing are complementary techniques for analysis and checking of properties of systems. Whereas verification aims at proving properties about systems by formal manipulation on a mathematical model of the system, testing is performed by exercising the real, executing implementation (or an executable simulation model). Verification can give certainty about satisfaction of a required property, but this certainty only applies to the model of the system: any verification is only as good as the validity of the model on which it is based. Testing, being based on observing a finite set of system behaviors, which for any realistic system is only a very small subset of all possible instances of system behavior, can never be complete: testing can only show the presence of errors, not their absence.

But since testing can be applied to the real implementation, it is useful in those cases where a valid and reliable verification model is not available. This can occur when the system is too large or too complex to make a complete verification model, when the system is a combination of formal parts and parts that are difficult to be formally modeled, e.g., physical devices, or an operating system, or when the structure or code of the system are not accessible, e.g., in case of proprietary code. Even when a model is available and formal verification is applied, testing is the technique to show that the model is valid, and that the real system conforms to the model.

For our simple purse application we have a formal specification, viz. the Statechart or Clean specification, we have the Java Card source code of the applet implementation, and, when necessary, we can have the Java Card byte code. This means that we could formally have verified the applet code in isolation with respect to the specification. We do not have the code of the Java Card execution platform: it is proprietary. This means that we cannot verify the execution platform, nor the combination of applet and execution platform. With testing we check the combination of applet, execution platform, and card hardware, i.e., we test whether the applet correctly communicates with the card environment.

Goal The purpose of this paper is to report about an experiment of automatic, formal specification-based testing of a Java Card applet. The aim of the experiment is to investigate the feasibility of formal testing, to show that indeed (inserted) errors can be detected in such an applet, and to compare with other testing approaches for smart cards.

The test set-up The applet that we test is the 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 applet is executed on the *CREF* Java Card simulation platform, which is part of the *Java Card Development Kit* [JC]. Using a simulation environment allows easy modification of the applet, e.g., seeding it with subtle bugs, whereas it does not restrict the generality of the experiment: all communication between the simulation environment and the test tool occurs via the standardized *ISO-7816* protocol [ISO97], so that the simulation environment could be replaced by a real smart card without any other changes.

To automate the testing process we use the test tool Gast [KATP03,KP04]. Gast is a test generation, -execution, and -analysis tool which takes properties expressed as functions in Clean, generates test input data based on the types occurring in those properties, sends these input data to the applet under test on the simulation platform, catches the return values from the applet, and checks whether these values satisfy the properties. The input to Gast is a Clean representation of the specification Statechart, and then all these steps are automatically performed in an "on-the-fly" fashion. "On-the-fly" means that test generation, -execution, and -analysis are alternatingly performed, only as far as needed, and that no explicit test case is generated.

An important practical aspect of automatic testing is connecting the test tool, i.c. Gast, to the system under test, i.c. the simulation platform running the applet. No general solution is available for implementing this part of the test tool, which is sometimes referred to as "glue software", or "test adapter". In our case it means that the *ISO-7816* and *TLP-224* protocols had to be implemented manually in the test tool.

Using all these ingredients a couple of test runs were executed, both with an (assumed to be) correct applet implementation, and with some error-seeded implementations.

Overview The case, the simple electronic purse application, and the test tool, Gast, are described in Sect. 2 and 3, respectively. Sect. 4 gives the formal specification of the electronic purse in the input formalism for Gast. Then the test architecture is described in Sect. 5: the set-up of components that allows tests to be performed. The tests and their results are presented in Sect. 6. Finally, Sect. 7 and 8 discuss related work, conclusions, and possible future extensions.

2 Case Study

To demonstrate our testing methodology we use a simple electronic purse application as a case study. This section describes the application. The basic events which 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` via `authenticate(pin)` before charging the card
- charge the card with an amount of `n` via `credit(n)`
- reset the card using a `puk` via `reset(puk)`

All these events are input events for the card, because they are triggered by the terminal, i.e., they are sent from the terminal to the card. To every input event, the card answers with a corresponding output event; these are:

– acknowledge an operation via `ack(n)`
– report an error via `error(n)`

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

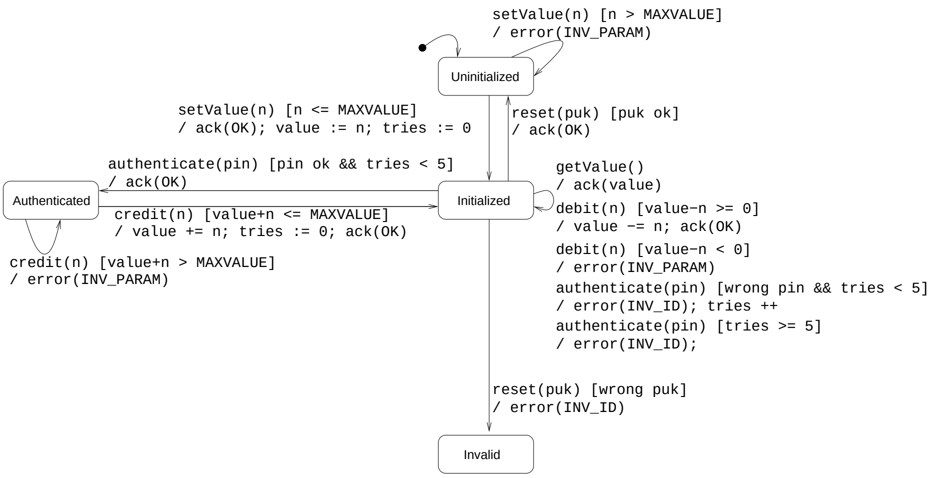
setValue(n) [n > MAXVALUE]
/ error(INV_PARAM)
Uninitialized
setValue(n) [n <= MAXVALUE]
/ ack(OK); value := n; tries := 0
reset(puk) [puk ok]
/ ack(OK)
authenticate(pin) [pin ok && tries < 5]
/ ack(OK)
Authenticated
Initialized
getValue()
/ ack(value)
debit(n) [value−n >= 0]
/ value −= n; ack(OK)
credit(n) [value+n <= MAXVALUE]
/ value += n; tries := 0; ack(OK)
debit(n) [value−n < 0]
/ error(INV_PARAM)
credit(n) [value+n > MAXVALUE]
/ error(INV_PARAM)
authenticate(pin) [wrong pin && tries < 5]
/ error(INV_ID); tries ++
authenticate(pin) [tries >= 5]
/ error(INV_ID);
reset(puk) [wrong puk]
/ error(INV_ID)
Invalid

Fig. 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:

$$\text{Authenticated} \xrightarrow{\text{credit(n) [value+n} \leq \text{MAXVALUE] / value} += \text{n; tries:=0; ack(OK)}} \text{Initialized}$$

Here, the input event i equals `credit(n)`. All events can pass values via variables, in this case the variable `n` is of type `unsigned short`[1], and represents the amount of money to be added to the card's value due to a charge operation. The actual value of the card is saved in the variable `value`. A transition can only fire when the corresponding guard `g` holds. Every card has a maximum value, represented by the constant `MAXVALUE`. The guard says, that one can only increase the value of the card by `n`, when the new amount `value+n` does not exceed `MAXVALUE`. Finally, `act` is a sequence of actions to be performed when the transition fires. At least one output event, here `ack(OK)`, must be performed. Additionally, variables can be set to new values. The actions are executed in the sequence they are written,

[1] the applet checks an `unsigned short` for being non-negative. We abstract from that in the statechart to keep it concise.

separated by semicolons. In this case, the variable `value` is firstly incremented by `n`, then the `tries` variable is reset to zero, and finally an acknowledgment is sent to the terminal.

At first, the card is in the `Uninitialized` state. It then 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. From the sixth wrong try on, one can no longer credit the card. The only solution is now to bring the card back to the credit institution and enter a reset code, called `puk`. If this code is entered 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 can not 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. Furthermore, a syntactically incorrect event that is not specified at all 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 actual state.

3 The Test Tool Gast in a Nutshell

3.1 Testing of functions with Gast

The test tool Gast performs automatic test generation, -execution, and -analysis [KATP03,KP04]. To do this, Gast basically has two functionalities: generating input data for the system under test, and checking properties specified to hold for the output data.

For the first functionality, Gast implements a generic algorithm to enumerate the elements of an arbitrary algebraic data type (ADT). Since the list of all elements of a recursive type is infinitely long, lazy evaluation is employed to generate only the fraction of this list that is actually needed. The elements are ordered from small, without recursive constructors, to large, with much recursion. Pseudo random numbers are used to make slight permutations in the list of elements. The unique feature is that Gast can do this for an arbitrary ADT; it does not need any further indication on how data is constructed.

This systematic generation of elements is secondly used to test specified properties stated in first order logic. In this paper we use a subset of first order formulas of the following form:

$$\psi = \forall x_1 \forall x_2 \ldots \forall x_n \; \varphi$$

with ψ having no free variables and φ being quantifier-free[2]. For the model of ψ must hold that the universes of the (typed) variables x_i are given as ADTs, and, of course, the functions must be computable. Both are guaranteed by the fact that Gast is implemented in the functional language Clean [PvE].

The formula ψ states, that for every possible valuation of the variables x_i the formula φ must hold. In the case of infinite universes we also have infinitely many valuations, therefore not all possible values can be tested (which would accord to a proof of ψ). Hence, only a finite subset of the valuations is tested. Due to the generation of values from small to large, this subset includes the common boundary values for recursive types.

If a valuation is found such that φ does not hold, it is proven that also ψ does not hold, and the testing can stop; the property is falsified and a counterexample has been found. Otherwise it stops when a given upper bound of test cases is reached. Then ψ is not proven, but at least the confidence in its validity has increased. This corresponds to the famous dictum by Dijkstra: *Testing can never demonstrate the absence of errors, only their presence.* For finite universes, however, Gast is able to prove properties.

As a simple example think of the presumably most famous ADT – a stack. Given a stack implementation Gast can be exerted to test properties like:

$$\forall\, i \,\forall\, s \,\big((\mathit{Top}(\mathit{Push}(i,s)) = i) \;\wedge\; (\mathit{Pop}(\mathit{Push}(i,s)) = s)\big)$$

or:

$$\forall s\big(\mathit{size}(s) \neq 0 \rightarrow \mathit{size}(s) > \mathit{size}(\mathit{pop}(s))\big)$$

with, for example, i being an integer, and s being a stack of integers. Gast will start creating stacks with increasing size in a way that avoids duplicates. For the integers, the pseudo random number generator is used after some predefined boundary tests, like -1, 0, 1, have been applied. If the implementation is correct, Gast will not find a counterexample, and will stop after the given number of test cases. Given a wrong implementation and a sufficiently high upper bound, a counter-example will be found.

The implementation of the stack may be written in the same language as in which the properties are stated, i.e., Clean, but the implementation can also be written in another language, to which Clean can connect, e.g., C. In this case, routines have to be added to convert the C stack to a Clean stack, and vice versa.

More details about Gast can be found in [KATP03,KP04].

[2] The use of existential quantifiers is, in principle, allowed, but leads in the case of infinite universes to semidecidability, which means that truth of an expression needs to be approximated sometimes.

3.2 State-Based testing with Gast

In order to test the Java applet the approach described above has limited power. It is possible to specify and test some specific properties, e.g., supplying the input `setValue(n)` in the initial state followed by a `getValue()` should yield an answer containing the same value `n`. It is, however, very tiresome to make a more or less complete set of tests in this way, which also takes into account the state behavior of the purse.

To allow for so called state-based testing, Gast has been extended to deal not only with first order properties, but also with specifications given as Extended Finite State Machines (EFSM). Such an EFSM comes quite close to the Statechart of Figure 1.

An EFSM consists of states with labeled transitions between them. A transition is of the form $s_1 \xrightarrow{i\ u^*} s_2$, where s_1, s_2 are states, i is an input which triggers the transition, and u^* is a, possibly empty, list of outputs. Like for the properties described above, the domains of the inputs, outputs, and states can be given by arbitrarily complex, recursive ADTs. This constitutes the main difference with traditional testing with EFSM's, see, e.g., [LY96], where for the testing algorithms to work all domains must be finite. Moreover, traditional algorithms require a deterministic EFSM, as opposed to Gast, and they usually do not consider lists of outputs including the empty one to mimic the absence of output, but this last restriction is, of course, trivially removed.

A Gast-EFSM is specified by a function that associates to each combination of current state and input a list of possible transitions. Each transition yields a pair of the list of outputs and the new state. When the list of transitions for a particular state and input contains more than one pair, the specification is nondeterministic. An implementation is then free to choose any of these specified pairs. If no transition is specified for a particular state and input the EFSM is underspecified, or incompletely specified. In this way, Gast is able to handle partial specifications. By specifying an EFSM as a function over arbitrary ADT's Gast avoids the state-space explosion problem, from which traditional (test) tools suffer, since these tools firstly transform any EFSM into an equivalent FSM by enumerating all, necessarily finite, data domains.

Inside Gast the generation of input sequences is separated from checking of the observed output behavior for these input sequences. This has the advantage that it is very easy to experiment with different strategies of generating input sequences. Some obvious choices for input generation are:

1. Since input sequences are just lists of elements, i.e., an ADT, Gast is able to generate these lists automatically and systematically. Before applying the next input symbol Gast checks whether it is appropriate in the current state of the specification. If not, the input is rejected, and testing continues with the next input sequence. This approach has as advantage that if an error pops up, one of the shortest paths to this error is found. This makes analysis of the error as easy as possible. For implementations that are slow compared to Gast, like the case study treated in this paper, Gast is able

to test only the input sequences that can be completely executed according to the specification.

2. If the specification is known to be an FSM Gast is able to generate input sequences that visit all transitions in the FSM. In order to check that the final state of each transition is correct, such a sequence is extended with a checking sequence, e.g., a Distinguishing- or UIO-sequence [LY96]. Such sequences must be supplied manually.
3. Instead of the standard generic generation of input values, the tester can manually define his/her own list of values. This can be used to check specific properties, or to exclude undesirable inputs. In the case study this is used to ensure that the `pin` used for authentication is correct in one of 8 situations, instead of having a chance of one out of 2^{32}. Another example is to focus on checking the applet's security by generating specific input sequences that consist of a `setValue` followed by a long list of `Authenticate`'s with random `pins`.
4. Finally, the specification can be used by Gast to generate long lists of inputs that correspond to valid transitions in the EFSM. This is the way to discover errors that pop up after a large number (e.g., 1000) of transitions.

In order to start each input sequence in the initial state it is required that the system under test (SUT) can always be brought back to its initial state. If this is not the case, Gast can easily be configured to construct paths starting in a different state than the initial one. Again, the generation stops when a given upper bound is reached. When a test case has passed the SUT, i.e., the observed outputs match the ones specified in the EFSM, the next test case is generated and applied. If a non-conformance is detected, the test is aborted, and the counter-example is reported.

4 The Purse Specification for Gast

Before starting to test with Gast, the Statechart specification of the simple electronic purse in Fig. 1 must be expressed as an EFSM in Clean, the input language for Gast. This section gives the details of this transformation. It can be skipped, without consequences for reading the remaining sections of the paper, by readers not interested in this transformation.

The Statechart specification of a Java Card application, like the one for the electronic purse, can be transformed directly into Clean following the principles explained in Sect. 3.2. We define a special ADT for the states, one ADT for the inputs, and an ADT for the outputs:

```
:: PurseState  = NotInitialized | Initialized Short Short
               | Authenticated Short | Invalid
:: PurseInput  = Reset Puk | SetValue Short | Debit Short
               | GetValue | Authenticate Pin | Credit Short
:: PurseOutput = Ack Short | AckOK
               | ErrorINV_PARAM | ErrorINV_ID | ErrorINV_CMD
```

The use of algebraic data types has several advantages. First, the use of ADT's enables us to write very concise specifications, and to have an almost one-to-one mapping to Statechart models. Secondly, the use a special algebraic data type for inputs enables GAST to systematically generate all possible sequences of inputs. Last but not least, the use of these types enables the CLEAN compiler to check that the specification is well typed.

The actual specification is the function `purse` that takes the current state and input as an argument and produces a list of tuples describing the allowed transitions. Each tuple contains the new state, like `NotInitialized`, and a list of outputs, like `[AckOK]`. The square brackets transform the constructor `AckOK` to the singleton list containing only this output. Since the specified applet is deterministic, there is no function alternative that contains a list of more than one tuple describing a new state and output.

CLEAN is a lay-out sensitive language. It does not use keywords for `then` and `else`, instead the then-part is placed on the next line, and the else-part on the line below.

```
purse :: PurseState PurseInput -> [(PurseState, [PurseOutput])]

purse NotInitialized (SetValue n)
    = if (n >= 0 && n <= MAXVALUE)
            [(Initialized 0 n, [AckOK])]
            [(NotInitialized, [ErrorINV_PARAM])]

purse (Initialized tries value) Reset
    = [(NotInitialized, [AckOK])]

purse (Initialized tries value) GetValue
    = [(Initialized tries value, [Ack value])]

purse (Initialized tries value) (Debit n)
    = if (n >= 0 && n <= value)
            [(Initialized tries (value - n), [AckOK])]
            [(Initialized tries value, [ErrorINV_PARAM])]

purse (Initialized tries value) (Authenticate pin)
    = if (tries < 5)
            (
              if (pin == PIN)
                    [(Authenticated value, [AckOK])]
                    [(Initialized (inc tries) value, [ErrorINV_ID])]
            )
            [(Initialized tries value, [ErrorINV_ID])]

purse (Authenticated value) (Credit n)
    = if (n >= 0 && n <= MAXVALUE - value)
            [(Initialized zero (value + n), [AckOK])]
            [(Authenticated value, [ErrorINV_PARAM])]
```

```
purse state any = []
```

The CLEAN code for the specification is very close to the Statechart model, leaving little room for errors. Only one of the issues encountered during the development and testing of the applet was due to a mistake in this transformation from the Statechart to the specification in CLEAN. Automating the transformation would help prevent such errors. Still, such a tool would require a handwritten formal specification, which might contain the same kind of mistakes as programming the specification in CLEAN.

The code for the specification as shown above only specifies how the applet should respond to correct inputs, and is therefore incomplete. Insiders can see this from the last line of the CLEAN code, which states that there are no transitions possible from any inputs on any state that are not handled by lines above the last. To make sure that the applet does not allow transitions (inopportune input events) that are not specified by the model, the CLEAN code can easily be made complete. Changing the last line of the code into

```
purse state any = [(state, [ErrorINV_CMD])]
```

specifies that the applet should output `error(INV_CMD)` for inputs on states that are not present in the model, and stay in the same state as it was before the input.

5 Testing Java Cards with GAST

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

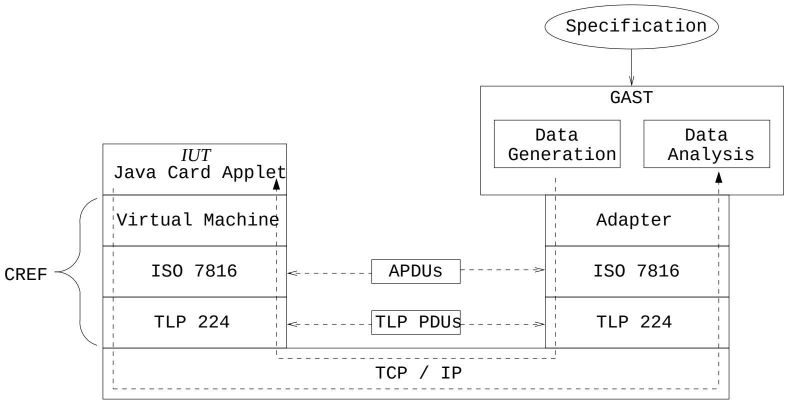

Fig. 2. The general testing framework.

The implementation under test, 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 was the *C-language Java Card Runtime Environment* (*CREF*), which comes with the *Java Card Development Kit* [JC]. *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. This communication uses special smart card protocols as specified by *ISO-7816-4* and *TLP-224* [ISO97], on top of a TCP/IP stack.

To communicate with the applet under test, GAST was enhanced to be able deal with these typical smart card communication protocols *ISO-7816-4* and *TLP-224*. On top of these protocol entities an *Adapter*, also referred to as "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 Sect. 4. Except for the access to TCP/IP, the right-hand side of Fig. 2 was 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, or testing a real smart card. The test architecture could easily be adapted to test real cards by exchanging only *CREF* with a real card and its reader. On the other hand, use of a simulation platform facilitates easy modification of the applet, e.g., for seeding errors, as we will see in the next section. Moreover, the use of a simulator has the advantage that once a test case brings the card in the `invalid` state, we can simply restart the simulation instead of reinstalling the applet. To do a lot of tests in the least amount of time, we deliberately avoided the transition to the `invalid` state with GAST.

6 Results

We tested the simple purse applet described in Sect. 2 with the test tool GAST in the test architecture of Fig. 2. Many test runs were performed, first during the development of the applet and its specification, and later with a couple of mutants. The results of these tests are described in this section.

During testing several differences between the applet and its specification did show up. Most differences could be accounted for by the applet, which did not always faithfully implement the model. But also the specification was changed on a few occasions, mainly because it was not clear, or not complete, on certain points.

6.1 Experiences during development

The formal model and the applet were simultaneously developed in an incremental way. Starting with a very trivial applet (to try out the framework described in the previous section), functionality was added to both the applet and the model until the semi-realistic purse application described in Sect. 2 was realized.

During this process GAST was applied to several incarnations of the applet and the model, yielding some interesting inconsistencies between applet and model. Some typical issues that were found are:

- Initially, class bytes and instruction bytes were chosen, for the actual inputs for the applet, that were not allowed according to the *ISO-7816* specification (yet these seem to work fine on actual Java Cards).
- Some checks for non-negativity, which are implicit in the model, and for maximum balance were not included in the applet.
- When the wrong pin-code was entered, the applet would internally change its state correctly, but fail to report this. Instead, a normal acknowledgment was sent back to the terminal.
- In the applet the `tries` counter was not set back to zero by the `reset` command.
- The model would increment the `tries` counter, even after it had reached 5. This caused an overflow in the model. This overflow was not present in the applet, otherwise we might have missed it completely.

These issues show that implementing a (relatively simple) Java Card applet in an incremental way is far from trivial. Having two different persons developing the applet and its specification, and having automatic testing for checking compliance between them, turns out to be very useful.

After some iterations, no more errors could be found by our testing tool. The resulting applet is considered to be bug-free. Running GAST on this final applet yielded the following result:

```
20,000,000 test paths used,
testing terminated,
19,316,244 rejected,
0 tests truncated,
in total 683,757 paths executed,
2,455,843 transitions.
```

The test paths above are the input sequences generated by GAST. Such a path is rejected if it cannot completely be executed by the specification. No inputs of a rejected path are applied to the applet, because the prefixes of the path that do successfully run through the specification have already been generated before by GAST. This is due to GAST's systematic generation of test paths from short to long. A test path is truncated when the next input of that path cannot be handled by the specification in the current state. This does not occur in our specification because it is deterministic.

Our testing tool ran for 1:45 h. on a 1400 megahertz PC. Only ten percent of its time was spent generating test paths and analyzing results. It appears that fifty percent of the time was used by Windows for communication between GAST and the simulator. The other forty percent is used by the simulator to run the applet. This strengthens our belief that GAST is an efficient test generation tool.

6.2 Mutant testing

The results given above show that applying GAST helps to find errors during development. A more systematic way to investigate the testing power of the GAST system uses *mutants*. Starting from the ideal (assumed to be correct) applet we inject typical programming errors into the applet, and analyze how long it takes for GAST to find these.

1. The first mutant differs from the ideal applet in the code dealing with the `setValue` command. The check on whether `value` is at most `MAXVALUE` is left out.
2. The second mutant differs from the ideal applet in the code dealing with the `credit` command. The comparison `n` > (`MAXVALUE`− `value`) is replaced with the comparison (`n` + `value`) > `MAXVALUE`. (The sub-expression `n` + `value` can overflow, leading to a negative balance, even if both `n` and `value` are positive.)
3. The third mutant differs from the ideal applet in the code dealing with the `debit` command. The check on whether the amount to be debited `n` is smaller than the actual balance `value` is left out.
4. The fourth mutant differs from the ideal applet in the code dealing with the `authenticate` command. The tries counter `tries` is not reset to zero, after successful authentication.
5. The fifth mutant differs from the ideal applet in the code dealing with the `reset` command. The tries counter `tries` is not reset to zero during a `reset`.

The results of the tests are in the table below. It shows the number of test events that took place, the number of test paths that were actually applied to the applet, the total number of generated paths by GAST, and the time it took to run the test. To speed-up the testing, not all generated test sequences are actually applied, as explained above: each path is checked against the CLEAN specification, and rejected if the specification returns an empty list of states.

mutant	test events	paths	generated	time	comments
1	166	33	173	0.4s	automatic
2	40676	7629	749701	71.0s	automatic
3	78	22	118	0.2s	automatic
4	1086	60	460	1.4s	guided
5	41704	6918	96386	66.0s	automatic

GAST was able to show the presence of errors in three mutants without any help. It needed guidance on the other two mutants, in order to find a counter example within a limit of 100000 generated tests. The counter examples found by GAST are listed in the table below.

mutant	counter example
1	`setValue(12567)`
2	`setValue(MAXVALUE), authenticate(pin), credit(32767)`
3	`setValue(0), debit(1)`

4	`setValue(0), authenticate(wrong pin),`
	`authenticate(wrong pin), authenticate(wrong pin),`
	`authenticate(wrong pin), authenticate(pin), credit(100),`
	`authenticate(wrong pin), getvalue(), authenticate(pin)`

5	`setValue(2436), authenticate(wrong pin), authenticate(pin)`

At first, it looked like GAST could not find the error in mutant 2. Even worse, a manually devised test path, `setValue(1), authenticate(pin), credit(32767), getvalue()`, did not find the possible overflow error present in the mutant. It turned out that GAST rejected the test path because, according to the model, `getvalue()` is not valid in the `Authenticated` state. This, in turn, is caused by the fact that GAST stays in the `Authenticated` state after the input `credit(32767)`, because `1 + 32767 > MAXVALUE`. Removing `getvalue()` from the devised test path allowed GAST to detect the error using the devised test path. Since many test paths are rejected by GAST because they contain inopportune input events, it becomes unlikely[3] that this specific test path would be one of the 100,000 generated test paths. We changed the generation of pin numbers to include the correct pin more often and added `MAXVALUE` and 32767 to the generated numbers, which also enabled GAST to find the error, without further guidance.

Although similar to mutant 5, GAST was not able to find the error in mutant 4. This seems to be caused by the sparseness of longer test paths within the given 100,000 limit. When we guided GAST towards the state were the error should show up, it did detect a difference between the model and the mutant. The similar error in mutant 5 was detected because the `reset()` transition is taken before executing every test path. Guiding GAST by adding the initial path `setValue(0), authenticate(wrong pin), authenticate(wrong pin), authenticate(wrong pin), authenticate(wrong pin), authenticate(pin), credit(1)` to all generated test paths did help.

When GAST would more often generate longer test paths, the probability of detecting the errors in mutants 2 and 4 would increase. But this would nullify a nice property of GAST: it always yields a short path that leads to the error. The latter feature did help a lot when investigating the differences between the model and the applet during development. Another option is to let GAST run longer and do more tests. We did not have the opportunity to run tests for days at a time, and, unfortunately, we do not have a good estimate of the expected time needed by GAST to find the errors.

[3] Note that numeric values and pin numbers are randomly chosen and change from one test run to the other.

Nonetheless, Gast was able to find counter examples for 4 out of 5 mutants in a very short time. It found errors in all of the mutants after some mutant specific guidance was given for mutant 4.

7 Related Work

In [dBM00] the authors use UML specifications, which are translated into Labeled Transition Systems to serve as input for the TGV tool [JJ02]. Instead of an on-the-fly execution, TGV needs an additional test purpose to generate test cases. This generation process is also named on-the-fly, but does not mean an on-the-fly execution of test cases like in our case. 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. The main difference to our approach is that we execute tests on-the-fly without the need for additional test purposes. This makes testing a lot easier and allows for a continuously running test execution which explores the (usually infinite) state space of the system under test (SUT). Still it is possible to guide the test case generation, as seen in the previous section. Another issue is that 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 add a symbolic treatment of data, the authors of [CJRZ01] use Input/Output Symbolic Transition Systems and the corresponding theory described in [RBJ00]. The basic approach is similar to TGV, hence also here test purposes are needed. The authors have not yet implemented a tool to automate their generation, but suggest a coverage based approach. The test automation is done via a translation to C++ code which is linked with the implementation. This restricts the SUT to be a C++ class with a compatible interface. Therefore testing real cards is not straightforward.

In [PPS$^+$03] the tool AutoFocus [HSSS96] is used to test smart cards. It models systems as a collection of communicating components. Instead of building on a formal conformance relation like TGV does, common testing strategies are used as a base for test case generation. They can basically be divided into functional, structural and statistical test case specifications. After the test cases are generated they are executed on a real card.

For a general introduction into these formal test tools, see [BFS04].

Rather than *testing* properties of the SUT, 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 Sect. 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 Fig. 1 can uniformly be translated to JML specifications as shown in [HOP03]. The resulting annotated Java Card applet can then be verified using one of the many JML tools [BCC$^+$03], for instance, the ESCJava2 static analyzer [CK04]. Most Java Card applets are small enough to even attempt a formal correctness-proof using the Loop tool, as demonstrated in [JOW04].

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. Such a specification can directly be translated into a corresponding Gast specification, and, in principle, there is no problem in completely automating this translation. Tests were completely automatically derived, executed, and analyzed. The feasibility of the approach was shown, and discrepancies between the formal specification and its Java Card implementation were successfully detected.

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 already 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 further insight on how to specify similar cases, and can henceforth serve as a pattern for these.

The tested mutants, representing certain fault models of typical programming errors, have increased our confidence in the error detecting power of the Gast algorithm. Nevertheless, we are planning to also use other test tools, e.g., the ioco-based tool TorX [Tre96,TB03], in order to compare, and to obtain insight in each tools' weak and strong points.

Another future goal is to extend the case study in different aspects, such as considering more complex applets, testing applets on real cards, and testing advanced aspects like the integration, interference, and feature interaction between different loaded applets on one card.

Finally, it will be interesting to 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 complementarity.

References

[BCC+03] 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, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[BFS04] Axel Belinfante, Lars Frantzen, and Christian Schallhart. *Model-based Testing of Reactive Systems - A Seminar Volume*, chapter Tools for Test Case Generation. LNCS. Springer Verlag, 2004. To appear.

[BGN$^+$04] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillman, and Margus Veanes. Towards a tool environment for model-based testing with AsmL. In A. Petrenko and A. Ulrich, editors, *FATES 2003 – Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2004.

[Che00] Z. Chen. *Java Card technology for smart cards: architecture and programmer's guide*. Addison-Wesley, June 2000.

[CJRZ01] 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*, pages 58–70, Cannes, France, September 2001.

[CK04] David Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2. Submitted for publication, 2004.

[dBM00] L. du Bousquet and H. Martin. Automatic test generation for Java-Card applets. In *4th Workshop on Tools for System Design and Verification*, July 2000.

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

[HOP03] Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. From finite state machines to provably correct java card applets. In Dimitris Gritzalis, Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sokratis K. Katsikas, editors, *Proceedings of the 18th IFIP Information Security Conference*, pages 465–470. Kluwer Academic Publishers, 2003.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AutoFocus - A Tool for Distributed Systems Specification. In *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, number LNCS 1135, pages 467 – 470. Springer Verlag, 1996.

[ISO97] ISO/IEC 7816-4, Information technology Identification cards Integrated circuit(s) cards with contacts Part 4: Interindustry commands for interchange. International Organization for Standardization (ISO), Geneva, CH, 1995/1997.

[JC] Java Card Technology. `http://java.sun.com/products/javacard`.

[JJ02] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. In *The Sixth World Conference on Integrated Design and Process Technology (IDPT'02)*, Pasadena, California, USA, June 2002. Society for Design and Process Science.

[JOW04] Bart Jacobs, Martijn Oostdijk, and Martijn Warnier. Source code verification of a secure payment applet. *JLAP*, 58:107–120, 2004.

[KATP03] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. In *Proceedings 14th International Workshop on the Implementation of Functional Languages, IFL 2002, Selected Papers, Madrid, Spain, September 16-18, 2002, Springer Verlag, LNCS 2670*, pages 84–100, 2003. see also `http://www.cs.kun.nl/~pieter/gentest/gentest.html`.

[KP04] Pieter Koopman and Rinus Plasmeijer. Testing reactive systems with gast. In *Proceedings Fourth symposium on Trends in Functional Programming, Edinburgh, Scotland, September 11-12, 2003.*, 2004. This is an improved version of technical report NIII-R0403, available from http://www.niii.kun.nl/research/reports/.

[LY96] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines – a survey. *Proc. IEEE*, 84(8):1090–1126, 1996.

[PPS$^+$03] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for smart cards. In *In Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, 2003.

[PvE] Rinus Plasmeijer and Marko van Eekelen. The Concurrent Clean Language Report, version 2.0. `http://www.cs.kun.nl/~clean`.

[RBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods – IFM 2000*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, 2000.

[TB03] 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, December 11-12 2003. 13 pages.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

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

[UML] UML resource page. `http://www.uml.org/`.