Model-Based MCDC Testing of Complex Decisions for the Java Card Applet Firewall

Roderick Bloem[1], Karin Greimel[2], Robert Koenighofer[1], Franz Roeck[1,2]

[1]Institute for Applied Information Processing and Communications
Graz University of Technology, A-8010 Graz, Austria
{roderick.bloem, robert.koenighofer, franz.roeck}@iaik.tugraz.at
[2]NXP Semiconductors Austria GmbH, Gratkorn, A-8101 Gratkorn, Austria
{karin.greimel, franz.roeck}@nxp.com

***Abstract*—Certification processes require the generation of models of a design. Using Model-Based Testing, these models can double as guides for test case generation. In this paper, we consider Boolean formulas that model a decision to be taken by a part of the software. We show how to use an SMT-solver to generate test cases that fulfill the MCDC coverage criteria on these models, in the presence of *strong coupling*. We show that the approach can improve test coverage, and finds a bug in an implementation of the Java Card Applet Firewall.**

***Keywords*—automatic test case generation; common criteria; java card applet firewall.**

I. Introduction

Certification of security critical embedded systems at a certain level requires that formal models of the design are created and verified against the security requirements. The main motivation for the work presented in this paper is to complement this certification effort with systematic testing. Reusing existing models for test case generation, we can increase the confidence in the quality of the actual implementation at little extra cost.

Common Criteria [5] is a typical, widely used certification scheme. It assures that *Security Functional Requirements* are met by the *Target of Evaluation*. It offers several *Evaluation Assurance Levels* (EAL). Starting with EAL6, a formal model is required to prove that the Security Functional Requirements are satisfied. For complexity reasons, this proof is (typically) carried out on the model and *not* on the actual implementation. We propose to complement the certification with test cases derived automatically from the model in order to close the link from the security functional requirements down to the actual implementation, as illustrated in Fig. 1. The arrow from the model to the implementation is dashed to emphasize that the implementation is often not derived from the model but developed independently. Thus, it is important to perform a conformance check, and test cases are a scalable and flexible option.

Models of security-critical systems often contain complex decisions, i.e., expressions evaluating to true or false. They may express, for instance, under which circumstances a user login should be successful or access to some resource should be allowed. Complex decisions may directly serve as models for stateless parts of the system (e.g., a method that checks if some access is allowed). They may also appear as guards in

This work was supported in part by the Austrian Research Promotion Agency (FWF) through project NewP@ss (835917).

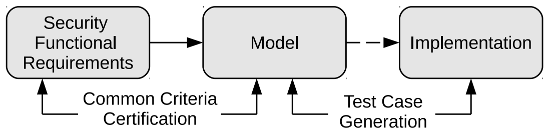

Security Functional Requirements
Model
Implementation
Common Criteria Certification
Test Case Generation

Fig. 1: Test case generation complements certification.

transition systems modeling stateful parts of the design. Such complex decisions are often difficult to test manually. First, there may be complex dependencies between the different parts of the decisions. Second, exhaustive testing is often infeasible, but we still want to cover the "interesting" cases. Test criteria help by defining which cases are interesting and have to be tested and we can use them to automatically generate test cases. The benefit of this Model-Based Testing approach is that the models are much simpler than the implementation, but precisely describe the various cases of interest. Also, the model acts as a test oracle. In our setting, the model is derived as a side-effect of the certification procedure, at no extra cost.

One widely used test criterion to measure code coverage is the Modified Condition Decision Coverage (MCDC) [9]. It is required by the US Federal Aviation Administration for safety critical software in aircrafts [13], and also used in many other domains. While MCDC is mostly used to measure the coverage of test cases with respect to the decisions in the implementation, we will apply the criterion to generate test cases from the decisions in the model.

In this paper, we show how to derive a test suite that achieves MCDC on a model that consists of logical decisions. Using a Satisfiability Modulo Theories (SMT) solver, we obtain values for the variables used in the decision. That is, our method not only computes the desired truth values of the different conditions (i.e., atoms) in a decision, but it also computes values for the (potentially non-Boolean) variables, which can then be used as test cases. It can handle complex interdependencies ("couplings") between the conditions by passing them on to the solver.

We apply our implementation in a case study of a Java Card applet firewall. This firewall is modeled as a decision under which access to an object is granted. Although a large set of manually constructed tests exists, the automatically constructed tests increase the code coverage. Using the model as a test oracle, our test suite also detects an inconsistency between the implementation and the specification due to an update of the reference manual which was not implemented.

Our test case generation method can be applied directly to testing Boolean effects in stateless systems. Our approach can also be combined with that of [19] to obtain test cases for systems that are modeled as transition systems, obtaining a test suite that exercises the guards of the state transitions.

Several papers on formal modeling for high assurance Common Criteria evaluations exist [4], [18], [7]. Both in [18] and in [7], the Java Card firewall is modeled and a theorem prover is used to prove that the model satisfies the access control policy. For a Common Criteria certification it is not necessary to formally link the model to the implementation. In [7], the gap to the implementation is closed by manual code-to-spec review. In contrast, we propose to close this gap by generating test cases from the formal model and run them on the implementation. In [16], the functional correctness of an OS kernel is directly proven for the implementation. This gives higher assurance of the correctness but increases the effort tremendously. Our approach of integrating formal verification at a high abstraction level and model-based testing at the implementation level gives a good trade-off between assurance and cost.

The idea of generating test cases based on formal specifications was already presented in [3]. Since then, a lot of research has been done in this field [21], [19], [14], [22]. In [14] a survey on testing with model checkers is given. Using model checkers, test cases are generated by defining trap properties in CTL formulas such that a counterexample represents a test case. The disadvantage of this approach, however, is that model checking can be quite resource intensive. Feeding the guards of the transitions into an SMT-solver, as we do, is potentially cheaper. The closest related work we are aware of regarding test case generation is presented in [19]. The authors compute test cases achieving MCDC on a specification by walking through the parse trees of the decisions. Depending on the logical operator they decide what the expression of the subtree should evaluate to. In contrast, our method (a) does not stop at the Boolean level but also produces values for non-Boolean variables appearing in the decisions, and (b) can handle complex dependencies between the different parts of the decision.

The rest of this paper is structured as follows. Section II introduces background and notation, and gives an example. Section III presents our quality assurance flow based on certification and test case generation. Section IV discusses our case study with the Java Card applet firewall, and Section V draws conclusions and gives ideas for future work.

II. Preliminaries

A. Decisions and Specifications

Let V be a set of variables ranging over a domain $\mathbb{D}$, and let F be a set of function symbols. A *term* over V and F is defined inductively as follows: (a) any variable $v \in V$ is a term, and (b) if $f \in F$ is a function symbol with arity n and $a_1, a_2, \dots a_n$ are terms, then $f(a_1, a_2, \dots a_n)$ is a term. For simplicity of the presentation, we assume that all variables have the same domain $\mathbb{D}$. E.g., $\mathbb{D}$ could be the domain of integers or bit-vectors of length 32. Also, all functions $f \in F$ are mappings $f : \mathbb{D} \times \ldots \times \mathbb{D} \to \mathbb{D}$. A *condition* is a function mapping a vector of terms to either true ($\top$) or false ($\bot$). A *decision* φ is defined inductively as follows: (a) every condition is a decision, and (b) if φ_1 and φ_2 are decisions, then $\neg\varphi_1$ and $\varphi_1 \vee \varphi_2$ are decisions as well. The Boolean operators $\neg$ and $\vee$ have their usual semantics. Other Boolean operators can be seen as shortcuts. A *specification* is a set $S = \{\varphi_1, \varphi_2, \dots\}$ of decisions.

We write $\mathsf{CoN}(\varphi) = \{c_1, c_2, \dots\}$ for the set of all condition nodes in the parse tree of decision φ, and $\varphi[c|\top]$ ($\varphi[c|\bot]$) for the decision φ with condition node $c \in \mathsf{CoN}(\varphi)$ replaced by $\top$ ($\bot$).

B. Test Cases and Specification Coverage

A *test case* for a specification $S = \{\varphi_1, \varphi_2, \dots\}$ is an assignment $t : V \to \mathbb{D}$ of values to all variables in V. We write $\varphi(t)$ or $c(t)$ to denote the truth value ($\top$ or $\bot$) of decision φ or condition node $c \in \mathsf{CoN}(\varphi)$ under assignment t. A *test suite* is a set $T = \{t_1, t_2, \dots\}$ of test cases.

Let φ be a decision, $c \in \mathsf{CoN}(\varphi)$ be a condition node, and $t : V \to \mathbb{D}$ be a test case. We say that c *determines* φ under t, written $\mathsf{det}(c, \varphi, t)$, iff $\varphi(t) \neq \varphi[c|\neg c(t)](t)$. That is, negating the truth value of c changes the truth value of φ.

Test suite T achieves *Masking Modified Condition Decision Coverage* [8] on specification S iff for all $\varphi \in S$:

$$\exists t, t' \in T : \varphi(t) \wedge \neg\varphi(t') \tag{1}$$

and

$$\begin{aligned} &\forall c \in \mathsf{CoN}(\varphi) : \exists t, t' \in T : \\ &\quad c(t) \wedge \neg c(t') \wedge \mathsf{det}(c, \varphi, t) \wedge \mathsf{det}(c, \varphi, t'). \end{aligned} \tag{2}$$

That is, every decision $\varphi \in S$ must evaluate to true and to false on some test. Also, every condition node c must evaluate to true and to false while determining the truth value of φ. Masking MCDC is also referred to as *Correlated Active Clause Coverage* in the literature [1].

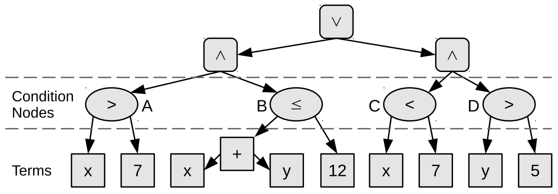
Unique cause MCDC [8] is a stricter variant. Whereas masking MCDC allows other occurring conditions to evaluate to different truth values for t and t' as long as the determination of c is preserved, unique cause MCDC requires the conditions to be same for both t and t'. Expressed more formally, we say that test suite T achieves unique cause MCDC on specification S iff for all $\varphi \in S$:

$$\exists t, t' \in T : \varphi(t) \wedge \neg\varphi(t') \tag{3}$$

and

$$\begin{aligned} &\forall c \in \mathsf{CoN}(\varphi) : \exists t, t' \in T : \\ &\quad c(t) \wedge \neg c(t') \wedge \mathsf{det}(c, \varphi, t) \wedge \mathsf{det}(c, \varphi, t') \wedge \\ &\quad \forall c' \in \{\mathsf{CoN}(\varphi) \backslash c\} : c'(t) = c'(t'). \end{aligned} \tag{4}$$

MCDC (either kind) can be calculated straightforward as long as all conditions are independent. However, variables may occur in more than one condition, and fixing the truth value of some conditions may determine others. If the truth value of one condition always flips when flipping the truth value of another condition, then these conditions are called *strongly coupled* [9]. If it flips in some but not in all cases, they are called *weakly coupled*. E.g., $(A > 5)$ and $(A < 9)$ are weakly coupled, whereas $(A > 5)$ and $(A \leq 5)$ are strongly



	x	y	A	B	C	D
t_1	7	6	0	0	0	1
t_2	0	13	0	0	1	1
t_3	7	0	0	1	0	0
t_4	0	0	0	1	1	0
t_5	0	6	0	1	1	1
t_6	8	5	1	0	0	0
t_7	8	6	1	0	0	1
t_8	8	0	1	1	0	0

Fig. 2: Example: Parse tree of a decision with test cases.

coupled. Due to coupled conditions, some test cases required for achieving MCDC may be infeasible.

Another metric, closer to exhaustive testing, is *Multiple Condition Coverage (MCC)*. MCC requires that for every decision, all possible combinations of the truth values of its conditions are tested. This results in 2^n test cases for n conditions in a single decision.

C. Example

Fig. 2 depicts the parse tree of the decision $\varphi = (x > 7 \wedge x + y \leq 12) \vee (x < 7 \wedge y > 5)$. Even though the decision is small, it is certainly not easy to test due to couplings between conditions. The parse tree contains four condition nodes, but from the $2^4 = 16$ possible truth value combinations, only 8 are satisfiable. They are listed as potential test cases t_1 to t_8 at the right side of the figure.

The four test cases t_2, t_4, t_7, t_8 achieve masking MCDC as follows. Condition node A is tested by t_4 and t_8: $\varphi(t_4) = \bot \neq \varphi(t_4)[A|\top] = \top$, so A determines φ under t_4. Analogously for t_8. In the same way, B is tested by t_7 and t_8, C by t_2 and t_7, and D by t_2 and t_4. However, these four test cases do not achieve unique cause MCDC because the pairs of test cases do not only flip the truth value of the tested condition node, but also others.

Unique cause MCDC can be achieved by the seven test cases $t_1, t_2, t_3, t_4, t_5, t_6, t_8$. Condition node A is now tested by t_3 and t_8. We have that $B(t_3) = B(t_8) = \top$, $C(t_3) = C(t_8) = \bot$, and $D(t_3) = D(t_8) = \bot$, so the truth value for the other condition nodes remains the same when testing A. In a similar way, condition node B is tested by t_6 and t_8, C by t_1 and t_2, and D by t_4 and t_5.

III. Certification with Test Case Generation

This section presents our proposed quality assurance flow, which is based on certification and automatic test case generation. It consists of the following four steps (see also Fig. 1):

1) Construct a model of the system.
2) Prove that the model of the system satisfies the requirements.
3) From the provenly correct model, automatically generate test cases.
4) Run the tests on the implementation.

Obviously, this flow does not give a formal proof of correctness for the implementation. It only proves that the model is correct with respect to the requirements. The test cases then verify that the model has been implemented correctly.

A. Certification

For certification under high assurance levels, like Common Criteria EAL6 or EAL7, formal models of the specification and of the design are required at different levels of detail, depending on the level of the certification. These models describe how the product implements certain parts of the specification. For Common Criteria, the requirements for formal security policy models are given in [6]. The ‘Bundesamt für Sicherheit in der Informationstechnik’, one of the certification bodies, published a guideline for the evaluation of security policy models [17]. The guideline suggests to use formal tools such as theorem provers or model checkers. We use model checking because these tools work fully automatically. A model checker takes a formal specification and a model as input. It returns `true` if the model satisfies the specification, giving a mathematical proof of correctness. Otherwise, if the model does not satisfy the specification, the model checker returns a counterexample [11].

We use the model checker NuSMV [10] for certification, as described in [4] for a smart card system. NuSMV has a proprietary modeling language, defining a finite state machine. A model consists of state and input variables, and of transitions defining how an input leads from one state to the next states. The rules on the transitions can be complex logical decisions. In the next section we will explain how to automatically generate test cases from complex logical decisions with high coverage.

B. Test Case Generation

We created a tool to compute a test suite T that achieves MCDC on a specification S. It uses the SMT-solver Z3 [12] to compute test cases as satisfying assignments of the constraints that have to be fulfilled by the tests. The decisions of the specification must be given in SMT-LIB2 format [2]. Our tool builds a parse tree of the decisions so that condition nodes can be found and replaced by $\top$ or $\bot$ easily. Next, it passes the constraints of Eq. 2 to the solver, one after the other, for all $\varphi \in S$ and $c \in \mathsf{CoN}(\varphi)$. For each satisfiable query, we can extract a pair of test cases t, t' as a satisfying assignment and add it to the test suite T. Variables which are irrelevant for the satisfying assignment will get a value which is either random or defined by the user. Unsatisfiable cases are reported to the user, because they usually indicate inconsistencies or redundancies in the decision. Before adding a new pair of test cases, we check if T already contains test cases that satisfy the constraints. This reduces the overall number of test cases. Finally, the test suite T is written into a simple text file, which can be parsed by a test adapter. Our tool supports masking MCDC, unique cause MCDC, and MCC.

IV. Case Study: Java Card Firewall

We applied our test case generation tool to the guard expressing the access rules in the formal model of the Java Card applet firewall, as specified in Section 6.2.8 of the JCRE specification, version 3 [20]. As a result, we obtain a set of test cases satisfying the MCDC criterion with respect to the guard in the model. Using MCDC ensures that each sub-item of the specification independently affects the access decision. After running the test suite it can then be assured, provided that none of the SMT-solver calls were unsatisfiable, that every

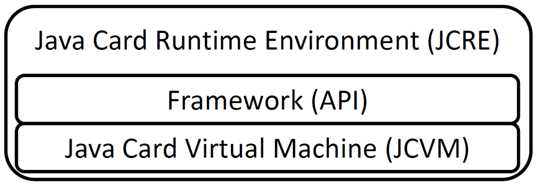

Java Card Runtime Environment (JCRE)
Framework (API)
Java Card Virtual Machine (JCVM)

Fig. 3: The Java Card Runtime Environment.

explicit requirement from the specification is implemented and evaluates, for the state which is used for testing, to the expected outcome. The generation of the test suite for the guard, consisting of 223 conditions, takes less than a minute and results in 205 test cases before removing duplicates and 127 test cases after the elimination. As some variables like the one to identify the current bytecode, the current context, the object owner and so on, occur in more than one condition, some SMT-solver calls were unsatisfiable due to coupled conditions.

A. Java Card Applet Firewall Model

Whereas in standard Java every applet runs on its own instance of a virtual machine, the Java Card virtual machine must be able to deal with several (independent) applets. The Java Card applet firewall ensures that applets cannot randomly access data belonging to other applets, but only in restricted cases. The applet firewall is part of the Java Card virtual machine (JCVM) (see Fig. 3) and checks every single access according to the JCRE specification [20].

Our model of the Java Card firewall (see Fig. 4) consists of only two states: `idle` and `locked`. As long as the firewall is in the `idle` state, it performs the required checks for the Java Card virtual machine. If an access is denied, a *SecurityException* is thrown. The JCVM then has to handle this exception, e.g. reset a started transaction, while the applet firewall doesn't do anything. This situation we have modeled by introducing the second state called `locked`. Conceptually, this behavior is very simple. The difficulty for testing the firewall stems from the very complicated decision when to allow access.

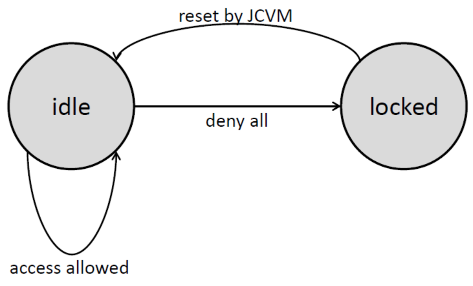


Fig. 4: Abstract model of the Java Card applet firewall.

The access rules are modeled by two transitions. The first transition, representing access allowed, is a self loop of the idle state. The second transition is a *deny all* transition, going from idle to locked with a lower priority. This realization ensures that every access, which is not explicitly allowed, is denied. The guard of the self loop is a formalization of Section 6.2.8 of the JCRE specification [20] such that a satisfying assignment for the formula corresponds to an access which is allowed.

Example 1: Section 6.2.8.7 of the JCRE specification [20] specifies access rules for the bytecode `athrow` by saying:

- "If the object is owned by an applet in the currently active context, access is allowed.
- Otherwise, if the object is designated a Java Card RE Entry Point Object, access is allowed.
- Otherwise, if the Java Card RE is the currently active context, access is allowed.
- Otherwise, access is denied."

This can be translated into a formula

$$\begin{aligned}&(\text{bytecode} = 7)\wedge\\&((\text{Owner} = \text{FLAG\_CurrentlyActiveContext})\vee\\&(\text{FLAG\_entryPointJCREObject})\vee\\&(\text{FLAG\_CurrentlyActiveContext} = 0)),\end{aligned} \tag{5}$$

where $(\text{bytecode} = 7)$ checks if the bytecode equals *athrow*, and the remaining three lines correspond to the first three bullets copied from the JCRE specification, with the constant 0 encoding the JCRE context. This example illustrates that formulating the specification of the firewall is (for the most part) rather straightforward.

B. Evaluation Setting

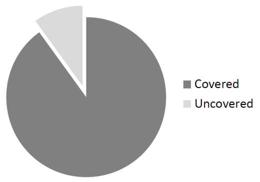
We compare the quality of the test cases created automatically using our tool to that of the JCTCK, a hand-crafted test suite. The hand-crafted tests are given as Java Card applets. They test the whole implementation of the Java Card runtime environment and not only parts of it. The test harness for these tests simply runs the applets. In contrast to that, our test adapter runs the test cases as module tests implemented in C, which is the language the Java Card operating system is programmed in. It sets up the memory as required from the test case and calls the relevant bytecode implementation which performs the necessary firewall checks. If access is denied, a security exception is thrown, otherwise access was allowed.

To evaluate the code coverage for both test suites, the functions belonging to the Java Card applet firewall were instrumented using a code coverage tool. The code was instrumented in a way such that covering all instrumentations corresponds to condition coverage plus basic block coverage. After running the test suites, analysis can be performed on the collected data and the code coverage can be compared.

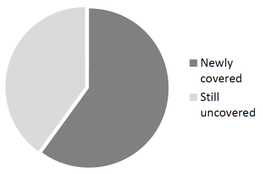
Of course, the test cases do not only have the purpose of covering as much code as possible, but also to check if the applet firewall's behavior conforms to the JCRE specification. The provided JCTCK installs and runs the applets and the applets themselves check if the result corresponds to the expected outcome. The test adapter for our automatically constructed test cases calculates the expected result, namely either access allowed or denied, by evaluating the decision in the model on the test case input data. If the access is denied, a security exception is thrown by the implementation

TABLE I: Instrumentations and achieved coverage

test suite	covered / total	percentage
JCTCK	64/71	90%
our test suite	63/71	89%
together	68/71	96%



(a) JCTCK coverage



(b) Additional coverage on the previously uncovered code

Fig. 5: Additional coverage on previously from the JCTCK uncovered code achieved by our test suite.

and caught by the test adapter. The test adapter finally checks for discrepancies between occurred and expected exceptions.

Our test approach is quite different from that of the JCTCK, so we are going to evaluate only results which are targeted from both test adapters. Moreover, we will explicitly mention if certain test goals can not be reached due to limitations stemming from the type of test. One such limitation is, for example, that a Java Card applet is restricted in its object creation.

C. Code Coverage Results

As we consider Section 6.2.8 of the JCRE specification as base for our test suite, we only evaluate the coverage in the functions of the applet firewall dealing with this Chapter. In some of the functions, the end can not be reached due to thrown exceptions. The total number of 78 instrumentations for the coverage is reduced by those, such that a coverage of all remaining 71 instrumentations corresponds to 100 % coverage. After running the test suites and storing the results for each test suite, the code coverage investigations reveal that neither of the test suites did achieve a full condition plus basic block coverage of 100 % (see Table I). However, when using both test suites it is possible to increase the coverage of the JCTCK by six percent, from 90 % to 96 %. This means that the automatically generated test suite covers 60 % of the cases that are missed by the JCTCK (see Fig. 5).

As there are only a few uncovered parts of the source code (see Table II), we will now explicitly discuss every one of them. The first condition that was not fully covered by our automatically generated test suite was a null pointer check. Some of the firewall functions perform a null pointer check

TABLE II: Conditions which were not fully covered

condition	JCTCK	our test suite
is object a null pointer	-	not to `true`
is object a global array	not to `true`	-
is object a shareable object	not to `false`	not to `false`
access of shareable object	not to `false`	not to `false`

before using the pointer. This condition is impossible for our test suite to cover, because our test suite is generated based on Section 6.2.8 of the JCRE specification and null pointers are not mentioned there. Therefore, no test case is generated targeting null pointers.

A check if the accessed object is a global array is covered by our automatically generated test suite but not by the JCTCK: The JCTCK was not able to make the condition in the source code evaluate to `true`. The reason is that the condition is disjuncted with a check if the object is a temporary entry point object in the source code. As there is only one global array in the Java Card implementation, namely the APDU buffer, and this one is also a temporary entry point object, the short circuit evaluation in the C semantics renders it impossible to let the global-array-check evaluate to `true`. In contrast to that, our test adapter sets up the memory as required without the restrictions for a Java Card applet and was therefore able to generate an object which is no temporary entry point object but a global array.

Neither of the two test suites was able to make the checks regarding shareable objects evaluate to `false`. This is due to implementation specifics, which perform a check if the class or interface is shareable already in the implementation of the bytecode itself before calling the actual firewall function. Therefore, the firewall function is only entered if the condition evaluates to `true`. Note that in the implementation the firewall is not a monolithic function isolated from the rest of the code, but rather an optimized implementation taking advantages of available code.

So, in summary, the only parts of the firewall which are not covered by both test suites taken together are conditions that can only evaluate to fixed truth values due to checks that are made elsewhere in the code. Beside these conditions, full condition and basic block coverage is achieved on the code relevant for Section 6.2.8 of the JCRE specification.

D. Error Detection Results

All tests from the JCTCK relevant for the Java Card applet firewall passed with success. From our automatically generated test suite, however, the result of three test cases did not match the outcome of the oracle. Two of those were false positives: The firewall didn't deny access to objects with some attribute combinations because other parts of the implementation ensure that these attribute combinations can never occur. The third failing test case detected an actual inconsistency. It tested an access rule from Section 6.2.8.9, namely "*Otherwise, if the object is designated a Java Card RE Entry Point Object, access is allowed*". The result was a Security Exception, whereas the oracle expected that the access would be allowed because of this rule. An inspection of previous versions of the specifications confirmed that this access rule was added to the specification in version 2.2 [15], however, a review of the source code showed, that this modification of the specification was not implemented. Due to the limitations for object creation via Java Card applets it was also not possible to create a test case for the JCTCK which tests this behavior on the system level, so this inconsistency remained undiscovered until now.

V. Conclusion and Future Work

In this paper, we have presented an automatic test case generation technique to achieve Modified Condition Decision Coverage on complex logical decisions. We implemented this approach in a test case generation tool using an SMT-solver to compute tests as satisfying assignments for the constraints that have to be satisfied to meet the coverage criterion. Our approach can handle complex couplings between different parts of the decision by delegating them to the SMT-solver. Our test case generation method can complement certification in the software development process by taking the existing models and closing the link from the requirements down to the implementation.

We evaluated our approach on an implementation of the Java Card operating system with focus on the applet firewall. Our tool produced only a small amount of test cases, but was able to improve the code coverage (condition + basic block coverage) of the existing test suite so that now all reachable locations and cases are covered. The additional tests produced by our tool also revealed that an update of the specification was not implemented. This confirms that different levels of testing are useful. System tests have to be complemented by unit and module tests because certain scenarios cannot (easily) be produced in the integrated system. The MCDC criterion proved to be effective in our setting because it tests the different parts of the decisions in isolation without producing too many test cases.

In the future, we plan to extend our test case generation approach and tool to deal with stateful models directly. This will relieve the user from writing a test adapter that brings the system into the desired state before the tests can be applied.

References

[1] P. Ammann, A. J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *International Symposium on Software Reliability Engineering (ISSRE'03)*. IEEE, 2003, pp. 99–107.

[2] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[3] G. Bernot, M. C. Gaudel, and B. Marre, "Software testing based on formal specifications: a theory and a tool," *Software Engineering Journal*, vol. 6, no. 6, 1991, pp. 387–405.

[4] G. Beuster and K. Greimel, "Formal security policy models for smart card evaluations," in *Annual ACM Symposium on Applied Computing (SAC'12)*. ACM, 2012, pp. 1640–1642.

[5] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 3 – Part 1: Introduction and general model*, July 2009.

[6] *Common Criteria for Information Technology Security Evaluation Version 3.1 Revision 3 – Part 3: Security assurance components*, July 2009.

[7] B. Chetali and Q. H. Nguyen, "Industrial use of formal methods for a high-level security evaluation," in *International Symposium on Formal Methods (FM'08)*, ser. LNCS, vol. 2404. Springer, 2008, pp. 198–213.

[8] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," DTIC Document, Tech. Rep., 2001.

[9] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, 1994, pp. 193–200.

[10] A. Cimatti *et al.*, "NuSMV version 2: An opensource tool for symbolic model checking," in *Computer-Aided Verification (CAV'02)*, ser. LNCS, vol. 2404. Springer, 2002, pp. 359–364.

[11] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 1999.

[12] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[13] R. T. C. for Aeronautics (RTCA), "RTCA-DO-178B: Software considerations in airbone systems and equipment certification," Dec. 1992.

[14] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, Sep. 2009, pp. 215–261. [Online]. Available: http://dx.doi.org/10.1002/stvr.v19:3

[15] S. M. Inc., "Java Card $^{\text{TM}}$ 2.2 Runtime Environment (JCRE) Specification," 2006.

[16] G. Klein *et al.*, "seL4: formal verification of an operating-system kernel," *Communications of the ACM*, vol. 53, no. 6, 2010, pp. 107–115.

[17] H. Mantel, W. Stephan, M. Ullmann, and R. Vogt, *Guideline for the Development and Evaluation of formal security policy models in the scope of ITSEC and Common Criteria Version 2.0*, December 2007.

[18] S. Motre and C. Teri, "Using B method to formalize the java card runtime security policy for a common criteria evaluation," in *National Information Systems Security Conference (NISSC'00)*, 2000.

[19] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software Testing, Verification and Reliability*, vol. 13, 2003, pp. 25–53.

[20] Oracle, "Java Card 3 Platform Runtime Environment Specification, Classic Edition Version 3.0.4," 2011.

[21] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31, no. 6, 1988, pp. 676–686.

[22] E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a boolean specification," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, 1994, pp. 353–363.