Electronic Notes in Theoretical Computer Science 80 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume80.html LastPage pages

Model-Based Test Case Generation for Smart Cards[1]

J. Philipps[a], A. Pretschner[a], O. Slotosch[a], E. Aiglstorfer[b], S. Kriebel[b], K. Scholl[b]

[a] *Validas AG, Lichtenbergstr. 8, 85748 Garching, Germany*[2]

[b] *Giesecke & Devrient GmbH, Prinzregentenstr. 159, 81667 München, Germany*[3]

Abstract

Testing denotes a set of activities that aim at discovering discrepancies between actual and intended behaviors of a system. Often, the intended behavior is known only implicitly, which renders the process of testing unstructured, unmotivated in its details, and barely reproducible. The use of explicit and executable models to describe the intended behavior promises to solve these problems. We use an industrial case study—a smart card application—to present a method for automatically generating test cases from such explicit models. The test cases are used both to validate the model and verify the actual card.

Key words: Test case generation, smart cards, modeling languages, CASE.

1 Introduction

Cost-effectively building the right system and building the system right continue to be the major problems of software and systems development. The right system is what customers desire; before development, it exists only in their minds. Unfortunately, these non-explicit requirements—because of their informal nature—are no suitable grounds for binding contracts. Specifications tend to be more formal and more precise; they try to capture the desired behavior of a system. The idea is that specifications serve as a blueprint for development and, once a system is built, as a reference to show that the system conforms to this specification, i.e., that it is built right. Two problems naturally arise: Specifications are often ambiguous, which, again, is because

[1] Support by the BMBF (project EMPRESS) is gratefully acknowledged.

[2] Email: {philipps,pretschner,slotosch}@validas.de

[3] Email: {Ernst.Aiglstorfer,Stefan.Kriebel,Kai.Scholl}@de.gi-de.com

they are not sufficiently precise. This is true even when systems development proceeds in close interaction with the customer. Also, it is not immediately obvious *how* to check conformance of a system with its specification.

To address these issues, we consider the use of explicit and executable models of a system. Assuming appropriate levels of abstraction, models are simplifications of the system, while still being precise and executable. Once a model is built and validated against the requirements (which, at one point or another necessarily are informal), it can serve as a system specification that is agreed upon. In principle, models can be used to automatically generate code; this is, however, not the subject of this paper. Rather, we focus on test generation, where the model serves as a reference implementation from which test sequences are generated.

For validation purposes, meaningful I/O sequences are generated. This requires not only the model of a system but also a test case specification. Roughly, the test case specification formalizes a given test purpose. A test purpose may be functional ("derive a test suite for testing a particular protocol"), structural ("derive a test suite that covers all branches"), or stochastic ("derive a test suite randomly or on the grounds of a given input data distribution"). Our test case generator then finds test sequences that satisfy the test case specification. These I/O sequences are execution traces of the model, and the validity of the outputs must be checked manually. An automated check could only be done against another model or formal specification of the system; clearly, this shifts the problem but does not solve it.

Once a model is considered valid, we use exactly the same test case generation technology to derive test suites for the actual system. The difference is that since the model is considered to be valid, we can automatically compare the system's output with the intended output as given by the traces of the model. This step requires adaptors that bridge the different levels of abstraction of system and model—after all, models are simplifications of the system and not mirror images.

We illustrate this approach with a model of the wireless application protocol identity module (WIM) for cellular phones [21]. Roughly, this module takes care of card holder verification (PINs and PUKs), of cryptographic operations such as computing digital signatures, enciphering and deciphering, and of the security related parts of the RSA handshake between the mobile equipment and some server.

The general approach is depicted in Fig. 1. In a nutshell, the model and a set of test case specifications are used to generate a test suite. This test suite is concretized in order to be applicable to the actual WIM. The model abstracts from card-specific data like PINs or keys, and this missing information is inserted from a separate data source. The concretized input is fed into the card, and the card's actual output is compared to the model's output that encodes the intended behavior.

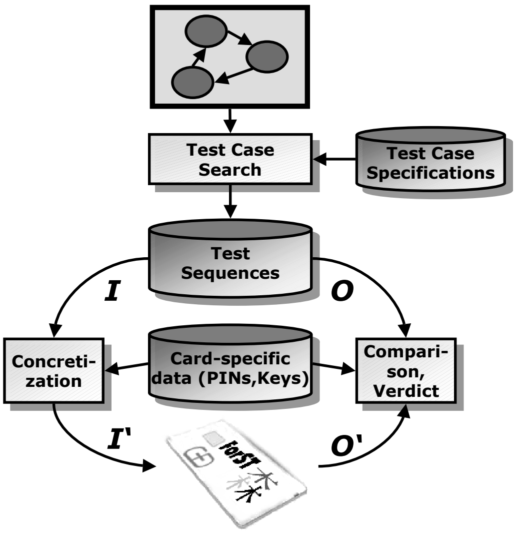

Fig. 1. Model-Based Testing

Contribution

The contribution of the paper is a proof of concept of the test case generation technology described in earlier work [16,14] and an extension of previous test generation work in the smart card field [17]. The WIM had undergone extensive testing at Giesecke & Devrient and is already being marketed; the goal of the work presented here is to provide evidence that the structured, automatic, and reproducible generation of test cases from models is possible and a viable alternative to current test design methods.

Outline

Section 2 describes the WIM and its model. In Section 3, we present our approach to test case generation on the grounds of symbolic execution. Some of the test case specifications we used for the WIM are exposed. Section 4 describes the test execution environment and shows how the abstract model and the concrete system can be related one to another. Section 5 provides an evaluation of the generated test cases. Related work is presented in Section 6, and Section 7 concludes.

2 The System and its Model

The system considered in this paper is the WAP identity module (WIM), which is used in the wireless access protocol extension of the GSM [7] standard for cellular phones to provide transport level security, digital signatures and in general public key cryptography.

The WIM is deployed as a smart card application. Smart cards are complete one-chip computers with a microprocessor, RAM (currently 256-4096 Bytes), EEPROM (2-16 KBytes), and ROM (8-64 KBytes) and a simple serial interface for communication with a terminal (ATM, cellular phone).

Often, a dedicated coprocessor for cryptographic operations is also present. Smart cards are fully programmable, and usually contain a dedicated operating system for standard functions like card holder verification (usually based on PINs—personal identity numbers) as well as a hierarchic file system. Smart cards are frequently employed to hide secret data like private keys, as in the case of the WIM.

The programming model of a smart card is essentially a command interpreter: The card processor reads commands from its input channel (which is connected to, say, a mobile phone), parses the command and its parameters, executes the command (which can change the data stored on the card), and transmits a response over its output channel. Successful execution of a command can depend on the previous command history (e.g. the user must be authenticated, keys must be set, files in the card file system must have been opened).

The test generation techniques presented below are based on the modeling languages of the CASE tool AutoFocus [9,19]. AutoFocus is a tool for developing graphical specifications of embedded systems based on concise description techniques—loosely related to the notations of UML-RT—and a simple, formally defined clock-synchronous semantics, which makes it rather well-suited for the command/response sequences of smart cards.

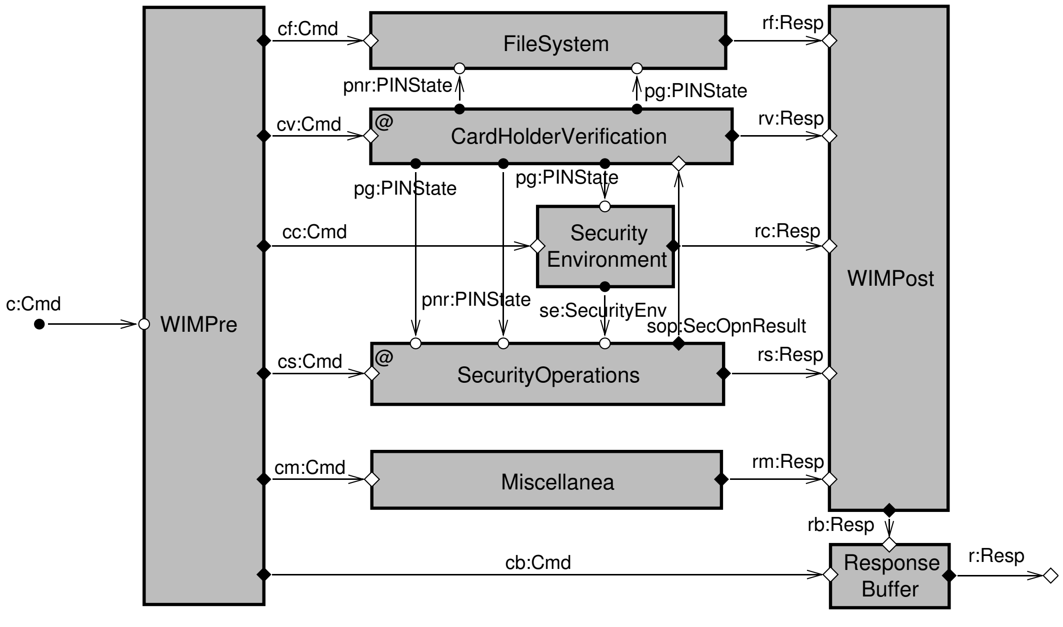

Fig. 2. WIM System Structure Diagram

The system structure is specified as a network of components that are connected by directed, typed communication channels. Fig. 2 shows a system structure diagram of the WIM. Commands enter the system at the left side. They are then classified and distributed to a number of functional blocks: file system, card holder verification (where the various PIN commands are handled), security environment (where keys, certificates and other parameters are stored), security operations (where the various cryptographic operations are performed). The responses of the functional blocks are gathered, in some cases buffered, and form the output of the WIM itself. The arrows between the functional blocks show the dataflow within the WIM.

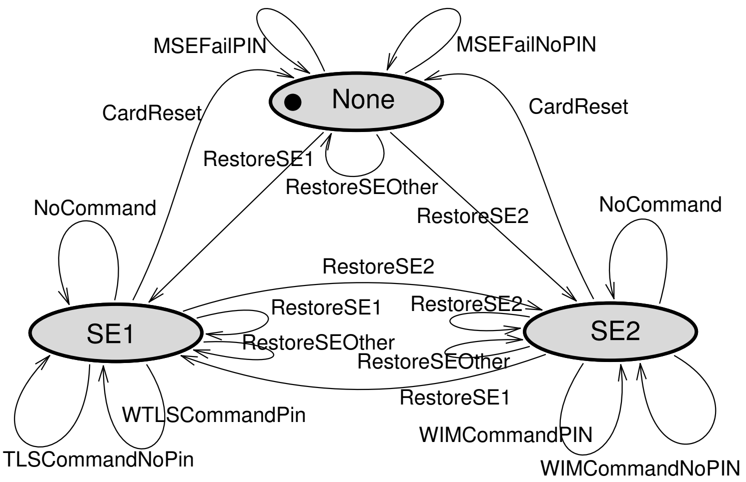

Fig. 3. State Transition Diagram

Component behavior is specified by state transition diagrams. Fig. 3 shows the state transition diagram of the security environment component, used to store keys and other cryptographic parameters for two different contexts (one for transport-level security operations, one for signature-related operations). In addition to the control states of the state transition diagram, components can have data states variables. Variable types, message types for the communication channels, classification predicates for messages as well as functions that transform the data state are specified in a simple programming language that is described in detail in [13].

The smart card architecture of Fig. 2 with its recursive structure of functional blocks with three-layers (command classification, command execution, response collection) is not particular to the WIM, but forms a reference architecture for arbitrary smart card applications. Moreover, hierarchy can be used not only to further decompose complex blocks (in the WIM, we decomposed the blocks for card holder verification and the cryptographic functions), but also to compose different card applications.

Not only the architecture is reusable: Certain card components, notably the file system and the card holder verification can be directly reused for models of other card applications. In fact, card models are better suited for reuse than the target code on the card itself, as there is no need for optimization in order to reduce code size.

3 Test Case Generation

To obtain test cases from the model, we require a test case specification. The choice of this specification is crucial, as it characterizes what is to be tested—it must reflect the notion of "good test cases" for a specific application. In addition, since testing itself incurs some cost, it is important to concentrate on a reasonable number of meaningful test sequences, and these test sequences have to be characterized.

We experimented with all three main classes of test case specifications—functional, structural, and stochastic. All classes can be reduced to a search problem in the computation tree of the model [14]. Functional specifications often involve finding sequences that produce a certain output: Scenarios, described themselves as partial I/O sequences can be broken into subsequences that fit into this pattern. A similar scheme applies to structural test case specifications where states are to be covered, transitions or sequences of transitions have to fire, etc. For stochastic test case specifications, a driver component is needed that generates inputs with given probabilities—the system's behavior between two generated inputs is then filled in by the test case generator.

Test Case Specifications

This paragraph describes the test case specifications we considered for the WIM.

Functional specifications. The WAP standards provide scenarios for some applications of the WIM, one of which exhibits the steps necessary to compute a digital signature. This scenario consists of (1) entering the correct PIN, (2) selecting the correct security environment, (3) setting the private key, and (4) performing the computation. Certain permutations of this protocol are legal in the sense that they also lead to a computation of the digital signature. The test case specification consists of a nondeterministic state machine, the driver, that encodes these legal permutations. In addition, for each state there are some commands that, according to the specification, do not change the status of the protocol, (e.g. selecting a file or asking for a random number), and some commands that can potentially affect the status of the protocol dependent on the card's response (e.g., changing the selected security environment). Both kinds of commands are easily encoded into the driver automaton (Fig. 4). This means that not only those traces are tested that eventually lead to a successful computation of the digital signature, but also those that issue the respective command and are rejected. Composed with the smart card model, the resulting system can be used to enumerate all traces of a given length.

Structural specifications. Because of the functional decomposition of the system we can generate tests for each component (functional block) more or less independently of the others (see [1] for a compositional approach to test generation). For instance, to test the component for card holder verification, one can decide to exclude file system commands in order to reduce the search

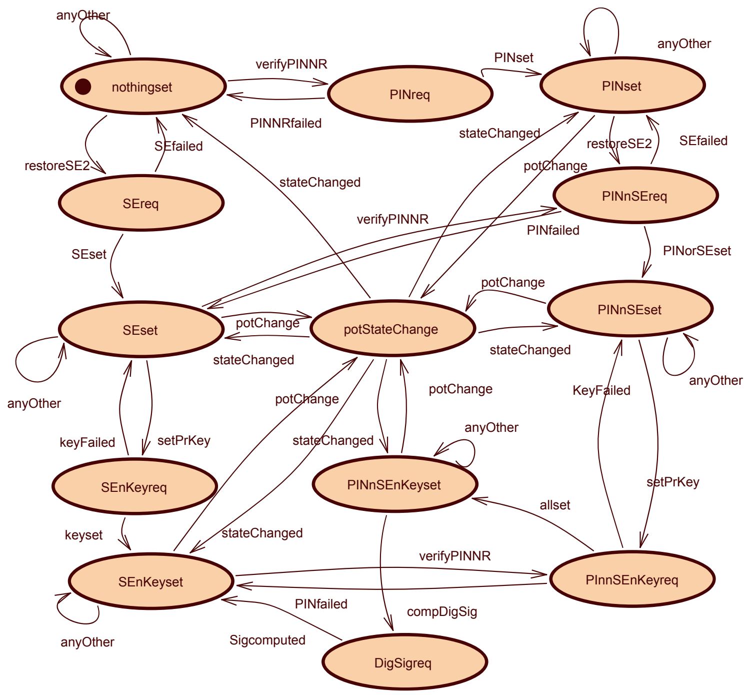

Fig. 4. Driver Automaton for Computation of a Digital Signature

space. Additional heuristics include admitting at most two changes of the security environment or at most one command that changes the state of a PIN to "Unverified" once it has been verified. The test case specification then consists of enumerating all traces up to a given length while certain commands, command combinations, or state combinations are prohibited.

We also used coverage criteria as test case specifications ([22] reviews coverage criteria to the end of both measuring the quality of a test suite and specifying test cases). It is relatively easy to lift these criteria to the level of models. We experimented with the modified condition/decision coverage (MC/DC) commonly used in the avionics field. Not surprisingly, we found that enumerating all sequences does yield a test suite that satisfies MC/DC—the reason is that we chose the maximum length to be large enough.

In order to document the relationship between a test sequence and the WIM requirements, we annotated the model with sections of the specification documents, and required all these annotations to be covered; this way we obtained requirements coverage. Conversely, we automatically structured the test suites generated by other test specifications w.r.t. the requirements they covered. This included both covering annotations and an automatically generated list of all possible command/response pairs.

Stochastic specifications. It has been observed that random testing is not inferior to partition testing w.r.t. error detection if no increased probability of errors can be assigned to certain input data partitions [4]. This motivates test case specifications that lead to the randomized generation of all (symbolic) test sequences up to a certain length (up to some hundred commands); to reduce the number of sequences generated, we demanded that two test generated sequences differed at least to a given extent.

Test Sequence Search

Searching the state space is achieved by symbolically executing the model. To this end, the AutoFocus model is translated into a constraint logic programming (CLP) language. The translation scheme takes advantage of the simple clock-synchronous semantics of AutoFocus; it is described in detail in [16,14].

In a nutshell, each transition of a bottom level component K—i.e., a component equipped with a state machine—is translated into a formula

$$step^K(\vec{\sigma}_{src}, \vec{\iota}, \vec{o}, \vec{\sigma}_{dst}) \Leftarrow guard(\vec{\iota}, \vec{\sigma}_{src}) \wedge assgmt(\vec{o}, \vec{\sigma}_{dst}) \tag{1}$$

indicating that given input $\vec{\iota}$, the component may proceed from control and data state $\vec{\sigma}_{src}$ to $\vec{\sigma}_{dst}$ by outputting $\vec{o}$, provided that the transition's guard holds true. The successor state is determined by the transition arrow's destination and an assignment that updates the component's data state. Components K that are not leaves of the component hierarchy and thus consist of subcomponents $k_1, \dots, k_n$ then recursively translate into

$$step^K(\vec{\sigma}^K_{src}, \vec{\iota}^K, \vec{o}^K, \vec{\sigma}^K_{dst}) \Leftarrow \bigwedge_{j=1}^{n} step^{k_j}(\vec{\sigma}^{k_j}_{src}, \vec{\iota}^{k_j}, \vec{o}^{k_j}, \vec{\sigma}^{k_j}_{dst}) \tag{2}$$

where internal channels, i.e., channels that connect subcomponents, are encoded as local variables of K and become parts of σ^K_{src} and σ^K_{dst}, respectively.

Running the resulting program with a logic programming engine then computes the set of all possible execution traces of the model. To reduce the size of this set, the traces are computed symbolically by means of constraints. For instance, if a transition guard requires that a value v is read from an input channel i, the CLP program does not enumerate all possible instantiations of i, but instead it creates two traces: one for the specific command, $i = v$, and one with a constraint specifying that the input must be different from that value: $i \neq v$.

Computing with sets of values rather than single values obviously helps to reduce the size of the state space. The following additional reduction is often useful but must be employed with care. Sometimes, one does not want to visit states twice. One can thus exclude already visited states (or short subsequences of such states) from the search process. Some care must be taken, however, to cope with the interplay of depth-first search and "prohibited" states [14].

Test case specifications are also translated into the CLP language and added to the program that represents the model. In addition to the methodical purpose described above, they serve three technical purposes: determining the end of the trace enumeration procedure, directing the search, and restricting the search space.

Before the symbolic execution traces found in this way can be used as test sequences, they must be instantiated. This can happen at random or by limit analysis. For example, the command “AskRandom” is used to deliver random bytes. Symbolic simulation yields an uninstantiated command “AskRandom(n)”, where n is a free variable. According to the range of n, the instantiation strategy binds this variable with values of 0, 1, 254, 255 and additional random values between 1 and 254. In general, this instantiation procedure takes place w.r.t. the above mentioned constraints that are collected during execution.

Note that we trust the card in the following sense. If the card returns a status that indicates there are n remaining attempts to verify a PIN, then we assume that there are indeed n remaining possible attempts. That is, we assume that the card response corresponds with the card’s internal state.

4 Test Execution

The test sequences generated according to the principles of the previous section link the WIM model with the WIM implementation on the smart card. For each command in each test sequence, the card response is compared to the response predicted by the model.

In the simplest case, if the observed card response differs from the expected response, the test fails. As usual, it must be checked whether the failure is caused by an error in the implementation, or by an error in the test case—in our case, by a modeling error due to a misunderstanding of the informal requirements.

Unfortunately, this simple comparison cannot be applied to all card commands. The WIM model is a simplification of the WIM application, and in particular does not contain a faithful description of the crypto algorithms used in the card. To cirumvent this problem, cryptographic command responses are not predicted by the WIM model, but rather by the test execution framework. Other commands, such as the random number generator, cannot be tested at all; here only simple consistency checks are performed during test execution.

Another difficulty in applying the generated test cases to the actual WIM stems from the different abstraction levels of card model and the card itself. Commands and responses in the model are described symbolically; they need to be translated into concrete byte strings. Conversely, card responses must be abstracted in order to be compared with the responses predicted by the card model for the command. This process is depicted in Fig. 5 for a card command which queries the number of remaining verification attempts.

Fig. 5 shows the simplest case of bridging the abstraction levels between model and smart card. Other card commands are treated somewhat differently:

- The model abstracts from the concrete contents of the smart card file sys-

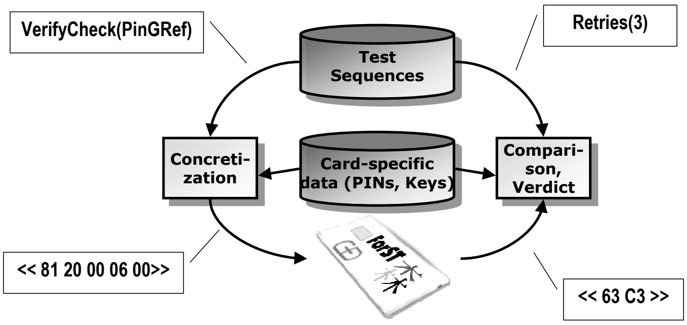

Fig. 5. Concretization and Abstraction

tem. Thus, the model responses of file access commands cannot be directly compared with the responses of the real card. Instead, only the expected length of the return data is compared.

- The command "AskRandom" already mentioned in Section 3 returns a certain number of bytes from a (cryptographically secure) random number generator. Obviously, there is no way for the model to predict these bytes; again, only the length of the response is checked.
- For cryptography functions, an abstract response is returned by the model which contains the necessary parameters (keys, certificates, the data to be encrypted, decrypted, signed or verified); it is left to the test framework outside of the model to verify the correctness of the card response with respect to these parameters.

Test execution itself is handled by a testing framework written in Python; this choice was motivated by the ease of accessing smart card communication libraries written in C and the support for integers of arbitrary length, which are essential for the implementation of the consistency checks of the cryptography operations of the WIM.

Given a set of test sequences, the test framework parses each sequence, sends each command in the sequence via a card terminal to the smart card, and checks consistency of the card's response with the expected response in the sequence; if there is a mismatch, the current sequence is abandoned and testing resumes with the next sequence. In addition, the framework has extensive support for logging, tracing and coverage measurements.

Card-specific data is described in simple configuration files, which relate concrete PINs, keys, certificates and other parameters with symbolic names used in the model.

5 Evaluation

With the test case generation techniques outlined in Section 3, we produced some 60,000 test sequences of varying length. To reduce test execution time, we randomly chose only 2-3% of the sequences; they take a comfortable hour to execute. Table 1 shows some qualitative results of this test set. The columns describe the test specification, the number of sequences used for testing, the average length of the sequences, the number of mismatches between model and card responses and the command coverage (the percentage of the command/response pairs occuring in the model that were indeed executed in the test sequences). Because of command redundancy in the different test sets, the coverage rate in the summary row is not the sum of the other rows. Note that command coverage does not exceed 93%. Closer examination of the model showed that the remaining pairs were linked to unreachable states in the card model; this implies that the test set was sufficiently large.

Specification	#Seq.	∅ lgth	Mism.	Cov.
Scenarios, mismatches	41	7	14	60%
Dig. sig. permutations	322	10	0	15%
Security environment	32	38	20	40%
Card holder verification	275	10	10	49%
Model cov. (Depth 5)	312	5	8	42%
Requirements coverage	528	7	32	63%
Summary	1506	8	84	93%

Table 1
Test Result Statistics

As seen in the table, testing revealed a number of mismatches between model and card responses. The mismatch rate is particularly high in the first row, as it contains a number of manual reproductions of mismatches for review purposes. All situations could be traced back to misinterpretations of the requirements documents, or in some inconsequential cases to code optimizations in the—experimental version—of the card software. This is not surprising: The project goal was to deliver a proof of concept for model-based testing, and the WIM card had undergone intensive testing before.

Nevertheless, examination of the generated test sequences has shown that they include the situations covered by the hand-written test sequences. The main area missing from the generated sequences are explicit injection of protocol errors, such as illegal length bytes, incomplete bit sequences, or timing errors.

Measurements on the card code itself showed that the generated test sequence set had an almost identical coverage to the test sequences (excluding the sequences dealing with the error situations mentioned above) designed for the commercial WIM version.

We do not give any further assessment of the quality of the generated test suite here. The reason is that it is most difficult to compare two test suites if the corresponding criteria are not based on syntactic properties such as code coverage. The main problem is that there is no general notion of the quality of a test suite.

It is straightforward to extend our technology to handle protocol errors as well. Even then the test case production cost can be expected to be substantially lower than for tests produced by the established non-model-based techniques.

6 Related Work

Test case generation on the grounds of explicit state transition diagrams is, among others, discussed in [3,18,5]. Our approach differs from that work in that we do not explicitly build the state space, but rather symbolically execute the system. Constraints for test case generation are also advocated in [11,12] the latter of which is also concerned with smart card testing. While conceptually similar, they seem not to use state storage strategies and the directed approach which we deem essential for the efficiency of symbolic execution. In the domain of processor validation [6,20], among others, use finite state machines—possibly automatically abstracted from VHDL code—to generate tests. These approaches do not profit from the advantages of symbolic execution with constraints. Symbolic execution for test case generation was widely discussed in the Seventies, e.g., [8], but the constraint solver technology available at that time was not sufficiently powerful.

Commonly, test cases are not generated based on models, but instead are written in a standard programming language, or in higher-level test case programming languages like TTCN [10]. One difference of our approach to protocol tests with TTCN is obviously our use of different abstraction levels, where part of the functionality is modeled, while other parts—notably the crypto commands—are handled in the test execution framework. In general, commands in our approach are more abstract than those for the implementation, and have to be translated during test execution. Another difference is our notion of test specification, which is more abstract and intensional than that of TTCN. Our hope is that by using generic test case specifications such as coverage of the model, of driver automata or of command/response requirements allows a higher degree of reuse over different card applications.

Testing, of course, is rather limited w.r.t. the properties that can be checked. It is, for example, doubtful that errors in the implementation of cryptographic algorithms can be found by testing alone; here correctness arguments will likely have to be based on core reviews and possibly formal verification. The Java Modeling Language [2] is an example of a behavioral interface specification language supported by a number of verification tools, which can be applied to smart cards implemented in the JavaCard language subset. It is also possible to generate test cases from JML specifications, but in general the use of a single artifact both for implemenation and as a test oracle is questionable at best.

7 Conclusion

Model-based test case generation has turned out to be a viable alternative to hand-written tests for smart card applications. With suitable test case specifications, it is easy to generate test sequences that cover the requirement specifications of the card commands, sequences that cover the model itself, sequences that permute common scenarios or taint them with additional commands, and sequences that explore limit cases of command parameters.

The use of models makes test generation itself cheap, but it involves additional cost in order to construct the model and the test case specifications. While there are a number of suitable notations for models (AutoFocus is but one possibility), for test case specifications we had to resort to ad hoc representations as CLP programs which encode both the test specification itself and some search heuristics. Current work aims to separate these two issues [14] and to allow composition of test specifications [15].

However, because of the high degree of reuse possible for both model (because of the standard architecture and standard functions blocks) and test case specification (because they describe rather abstract test goals), we expect most of these costs to amortize quickly.

Moreover, building models of a smart card application has intrinsic value: The need to abstract the textual requirements and to transform them into a deterministic, complete and executable model leads to an early clarification of the requirements. This clarification is quite important in practice: For the WIM, the requirements in the standard documents contained a number of contradictions and ambiguities; we expect this to be no different for other card applications. Obviously, the development of the card software itself can also benefit from these clarifications, and indeed models can serve as reference also for the software implementation. In principle, coding could even be automated by code generators, but then one must be careful not to generate code and test cases from the same model, as then—apart from possible errors in compilers or the code generator—the test cases will trivially be satisfied.

It remains to be seen whether the benefits of model-based testing can also be realized in other domains—automotive or avionics control systems immediately come to mind, as in their domain the use of modeling languages is already comparatively wide-spread.

References

[1] Bender, K., M. Broy, I. Péter, A. Pretschner and T. Stauner, *Model based development of hybrid systems: specification, simulation, test case generation*, in: *Modelling, Analysis and Design of Hybrid Systems*, LNCIS **279** (2002), pp. 37–52.

[2] Burdy, L., Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino and E. Poll, *An overview of JML tools and applications*, Technical report, Department of Computer Science, University of Nijmegen (2003), research Report NIII-R0309.

[3] du Bousquet, L., F. Ouabdesselam, I. Parissis, J.-L. Richier and N. Zuanon, *Specification-based Testing of Synchronous Software*, in: *Proc. 5th Intl. Workshop on Formal Methods for Industrial Critical Systems*, 2000.

[4] Duran, J. and S. Ntafos, *An Evaluation of Random Testing*, IEEE TSE **SE-10** (1984), pp. 438–444.

[5] Fernandez, J.-C., C. Jard, T. Jéron and C. Viho, *Using on-the-fly verification techniques for the generation of test suites*, in: *Proc. 8th Intl. Conf. on Computer-Aided Verification*, 1996.

[6] Fournier, L., A. Koyfman and M. Levinger, *Developing an Architecture Validation Suite—Application to the PowerPC Architecture*, in: *Proc. 36th ACM Design Automation Conf.*, 1999, pp. 189–194.

[7] *GSM 11.11*, Digital cellular telecommunications systems (Phase2+); Specification of the Subscriber Identity Module — Mobile Equipment (SIM-ME) interface.

[8] Howden, W., *Symbolic Testing and the DISSECT Symbolic Evaluation System*, IEEE TSE **SE-3** (1977), pp. 266–278.

[9] Huber, F., B. Schätz, A. Schmidt and K. Spies, *AutoFocus—a tool for distributed systems specification*, in: *FTRTFT'96, LNCS 1135*, 1996.

[10] ITU-T, "Recommendation Z.140, Testing and Test Control Notation version 3 (TTCN-3): Core Language," ITU, 2003.

[11] Legeard, B. and F. Peureux, *Génération de séquences de tests à partir d'une spécification B en PLC ensembliste*, in: *Proc. Approches Formelles dans l'Assistance au Développement de Logiciels*, 2001, pp. 113–130.

[12] Marre, B. and A. Arnould, *Test Sequence Generation from Lustre Descriptions: GATEL*, in: *15th IEEE Intl. Conf on Automated Software Engineering (ASE'00)*, 2000.

[13] Philipps, J. and O. Slotosch, *The quest for correct systems: Model checking of diagrams and datatypes*, in: *APSEC'99* (1999), pp. 449–458.

[14] Pretschner, A., *Classical search strategies for test case generation with constraint logic programming*, in: *Formal Approaches to Testing of Software (FATES'01)*, 2001, pp. 47–60.

[15] Pretschner, A., *Compositional generation of MC/DC integration test suites*, in: *Proc. TACoS'03*, Electronic Notes in Theoretical Computer Science **6**, 2003, pp. 1–11.

[16] Pretschner, A., H. Lötzbeyer and J. Philipps, *Model Based Testing in Evolutionary Software Development*, in: *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping*, 2001, pp. 155–160.

[17] Pretschner, A., O. Slotosch, H. Lötzbeyer, E. Aiglstorfer and S. Kriebel, *Model based testing for real: The inhouse card case study*, in: *Proc. 6th Intl. Workshop on Formal Methods for Industrial Critical Systems*, 2001, pp. 79–94.

[18] Raymond, P., D. Weber, X. Nicollin and N. Halbwachs, *Automatic testing of reactive systems*, in: *Proc. 19th IEEE Real-Time Systems Symposium*, 1998.

[19] Schätz, B., A. Pretschner, F. Huber and J. Philipps, *Model-based development*, Technical Report TUM-I0204, Institut für Informatik, Technische Universität München (2002).

[20] Shen, J. and J. Abraham, *An RTL Abstraction Technique for Processor Micorarchitecture Validation and Test Generation*, J. Electronic Testing: Theory&Application **16** (1999), pp. 67–81.

[21] WAP Forum, *Wireless Identity Module. Part: Security* (2001), wireless Application Protocol WAP-260-WIM-20010712-a.

[22] Zhu, H., P. Hall and J. May, *Software Unit Test Coverage and Adequacy*, ACM Computing Surveys **29** (1997), pp. 366–427.