Automated Test and Oracle Generation for Smart-Card Applications*

Duncan Clarke, Thierry Jéron, Vlad Rusu and Elena Zinovieva
{dclarke|jeron|rusu|lenaz}@irisa.fr

IRISA/INRIA Rennes, Campus de Beaulieu, Rennes, France

Abstract. We present work we are engaged in to develop symbolic test generation techniques and apply those techniques to testing of smart card applications. Beginning with (1) a system specification and (2) a test purpose expressed as symbolic labelled-transition-systems, we automatically derive tests to check conformance of an implementation to the behaviors of the specification selected by the test purpose. We present an example taken from a case-study we are developing based on the application of these techniques to the CEPS e-purse specifications.

1 Introduction

After decades of research into techniques for formal system specification, test generation, notions of test coverage, and methods for automating the evaluation of test results, the practice of software testing remains largely unchanged for most software developers. A developer familiar with the informal requirements for an application designs test cases in an ad-hoc fashion, runs those tests against an implementation, and compares the outputs of the implementation to his or her intuition regarding what the requirements dictate for the associated test case. Human-centric testing methodologies like this (1) are prone to error, (2) make poor use of existing formal specifications, and (3) make poor use of human resources.

Indeed, researchers and developers working in some specific problem domains have noted these facts, and taken steps to establish development methods with a sound formal basis that are open to automation. One of the most successful such efforts to date is in specification and conformance testing of protocols [ISO92]. In protocol specification and conformance testing a formal model of the protocol is created using a well-defined formalism with a sound mathematical basis such as SDL [IT94]

* Supported in part by DYADE action FormalCard, a joint project of INRIA and Bull/CP-8.

or LOTOS [ISO88], a conformance criterion [Tre94] is defined to establish exactly what it means for an implementation to conform to its specification, and a set of tests can be generated that have the ability to check conformance by driving a system under test through a desired sequence of states and comparing replies received to expected values. These techniques are developed to the point that several tools for the automatic creation of such test suites are available, including TorX [BFdV$^+$99], TGV [JM99] and TestComposer [KJG99].

Our work is an attempt to leverage the ideas underlying protocol conformance testing and high-efficiency test generation as embodied in the TGV tool, to automate the generation of tests for a more general class of applications. TGV, like most existing formal analysis tools, performs its analysis by enumerating the specification's state space. This leads to two significant problems when generating tests for large-scale systems: (1) state space explosion, as the variables in the specification are instantiated with all of their possible values, and (2) tests that are not readily understandable by humans when represented by a large network of states and transitions in a pure labelled transition system. To avoid these problems we are applying symbolic techniques to perform our analysis.

In [RdBJ00] the authors presented a method for the generation of symbolic test cases from system specifications and test purposes expressed in the Input/Output Symbolic Transition System (IOSTS) formalism. The models used are symbolic in the sense that items represented by variables over data domains in the control and computation steps of the specification remain variables in the generated tests. At no point in the analysis is it necessary to enumerate the state space of the specifications, test purposes or generated tests.

In addition to the elimination of the need for state space enumeration, the principal benefits of this approach are four-fold: (1) The derivation of tests *and* oracles from formal, operational specifications can be fully automated; (2) The tests are symbolic in the variables and parameters of the specification, so a single test can be generated and applied to implementations based on different specification parameter values; (3) The resulting tests are concrete, in the sense that once parameters are instantiated the tests can be translated easily to a test language and applied directly to real implementations (provided the model and the system under test are compatible at the interface level); and (4) the theory underlying the derivation of tests guarantees certain desirable properties, such as correctness and completeness of verdicts relative to the given test purpose.

Recent work by Martin and du Bousquet shares our goal of automation of test generation for smart-card applications. In [MdB00] they describe a testing methodology for smart card applications based on UML specifications, the use of TGV for on-the-fly test generation, and translation of TGV-generated tests to Java code for the creation of executable tests. There are three important points of contrast between their approach and ours: (1) They present their work using high-level UML specification models; at present our methods are based on the low-level IOSTS formalism, which will ultimately be used as an intermediate representation format. (2) The use of TGV for test generation introduces a risk of state space explosion if there are large data domains for inputs of the tester; the technique we describe is all symbolic, so there is no risk of state space explosion. And (3), TGV requires parameter values to be bound to individual values before test generation; the technique we describe allows system parameters to be bound to values as late as test execution time.

The remainder of this paper is structured as follows: Section 2 briefly describes the IOSTS formalism and an overview of symbolic test generation. Section 3 presents a brief example demonstrating the application of symbolic test generation to one feature described in the CEPS [CEP00] e-purse specifications. Section 4 describes the tool we are developing to automate symbolic test generation from formal specifications down to executable C++ test code. Section 5 closes the paper with summary remarks.

2 Symbolic Test Generation

Test generation is a program-synthesis problem: starting from the formal specification of a system under test and from a test purpose describing a set of behaviors to be tested, compute a reactive program that observes an implementation of the system to detect non-conformant behavior, while trying to control the implementation towards satisfying the test purpose. In this section we briefly describe our approach [RdBJ00] for generating symbolic test cases in the form of extended input-output automata.

The model. IOSTS (Input-Output Symbolic Transition Systems) are a model for reactive programs with symbolic processing of variables, parameters, and inter-process value passing. Several examples of IOSTS are given in the figures of Section 3. The syntax and semantics are formally defined in [RdBJ00]. We use IOSTS for describing specifications, test purposes, and test cases, and assume nothing about the black-box implementation other than that it is interface-compatible with the specification. Furthermore, we restrict our attention here to specifications and implementations that are *command-response*, meaning that an input, or sequence of inputs, may only be followed by at most one output (*i.e.*, a command is followed by at most one response), and there are no infinite loops of internal actions.

The IOSTS notation is rather low-level and is mainly convenient for processing by machine as done in symbolic test generation. For specifying systems at the user level, we plan to use a higher-level language which translates automatically to IOSTS (cf. Figure 1). The part of the diagram within the dashed box corresponds to the current status of the test generation tool. We also plan to incorporate mechanisms to automatically compute test purposes from, e.g., coverage criteria [RW85].

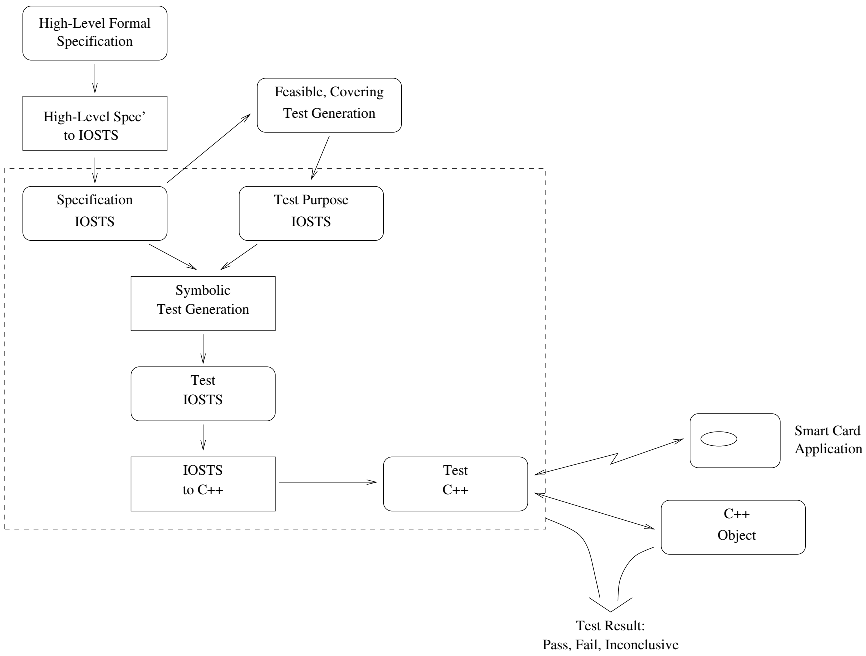

Fig. 1. Symbolic Test Generation Process.

The process. Symbolic test generation consists in computing, from specification and test purpose, a test case that covers all the behaviors of the intersection of these two elements. Then, the test case is translated into executable code to be run on the black-box implementation. Test generation consists of the following steps.

- *product* between test purpose and specification. This allows the selection of a subgraph of the specification that formally leads to the satisfaction of the test purpose. That is, only the subgraph leading to accepting states of the test purpose is kept. Here "formally" refers to the fact that symbolic variables and parameters are not interpreted, thus, actual reachability of the accepting states is not guaranteed. This problem is addressed in the simplification step.
- *closure* and *determinization* of the product. This operation attempts, through a set of heuristics, to produce a trace-equivalent system that has no (or fewer) internal actions and is deterministic. The heuristics will successfully terminate when applied to command-response systems.
- *adding verdicts.* This step consists in labeling some states with verdicts. "Pass" means that the test purpose was satisfied and no errors were detected (i.e., there were no observable differences between implementation and specification). "Inconclusive" means that, although no errors were detected, uncontrollable behavior has lead the test execution to a point where the test purpose cannot be satisfied any more. "Fail" means that an error was detected.
- *simplification.* At the end of the previous step, we have an IOSTS test case that is ready to be translated into a programming language. However, since there are potentially unreachable parts in the test case, we attempt to prune infeasible transitions by statically evaluating their guards using the Omega library [KMP$^+$]. The guards are analyzed to determine whether there is any valuation of variables that could satisfy the guard at run-time. If not the transition is removed along with any locations and transitions thereby made unreachable. This step is not mandatory, however, it may significantly reduce the size of the executable test case.
- *translation to C++.* This final step builds a C++ program from the IOSTS test case obtained after all previous steps. The Omega library [KMP$^+$] is called whenever there is a need to compute values to instantiate a symbolic output. The program is then ready to be compiled and linked with the implementation for test case execution.

Correctness of test cases. In [RdBJ00] we prove that the generated IOSTS test cases have, by construction, a set of correctness properties, meaning essentially that they do not produce false "Pass", "Fail" and "Inconclusive" verdicts. Furthermore, every test case generated from a given specification and test purpose has a *relative completeness* property, meaning that a "Fail" verdict will be produced in every circumstance where the implementation exhibits a non-conformance with the specification in a behavior targeted by the test purpose.

3 Example

In this section we present a brief example of symbolic test generation based on a feature of the CEPS e-purse specifications. CEPS (Common Electronic Purse Specifications) is a standard for creating inter-operable multi-currency smart card e-purse systems. The feature that we will generate tests for is the "CEP Inquiry - Slot Information" specified in Section 8.7.1 of the CEPS technical specifications[CEP00]. This feature provides a means for iterating through the slots, where each slot corresponds to one currency and its respective balance. The number of currency slots in the purse is implementation dependent, and the number of slots actually loaded with currency can vary during use, so a generic iteration scheme is provided for viewing each slot in turn. The specification requires that each currency be returned exactly once followed by an end-of-data marker. The order in which currencies are returned is unspecified and implementation dependent.

As described in Section 2, the process of test generation consists of (1) creating an IOSTS specification, (2) creating an IOSTS test purpose, and (3) generating an IOSTS test. We begin with the specification.

The specifications. Figure 2 presents the part of the IOSTS model of the CEPS specifications that describes iterative extraction of slot information from the card's point of view. The IOSTS is made up of locations and transitions. Transitions are decorated with their guard expressions, their input or output action (preceded by a "sync" keyword), and their assignments (using the ":=" notation for assignment). A transition is firable when its guard is satisfied and a complementary action is offered by the environment for synchronization. Transitions labelled with action "tau" are internal actions that can fire without synchronizing with the environment.

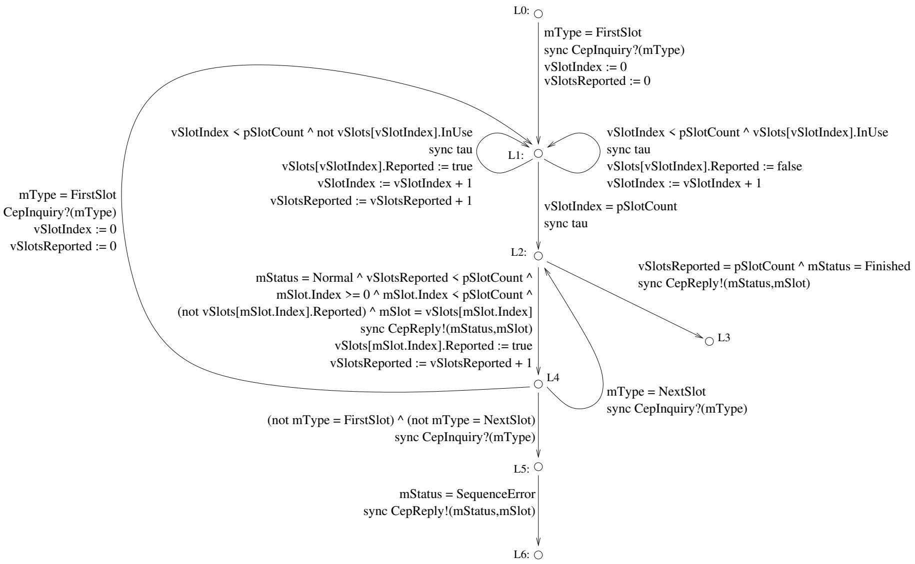

Fig. 2. Specification

The terminal to card interface for CEPS is based on command/response pairs, so the interaction between terminal and card begins with a CepInquiry command with a parameter mType equal to FirstSlot (*i.e.,* the transition from L0 to L1). After receiving this command the application performs internal computations to initialize the array of slot information (vSlots), marking slots that are in use as ready for reporting (*i.e.,* vSlots[...].Reported := false). This array has parametric size specified as pSlotCount. The initialization is in essence a "for" loop on variable vSlotIndex from zero to pSlotCount minus one at location L1.

When vSlotIndex reaches pSlotCount, an internal action is executed, taking the application to location L2 where it is ready to send its first reply and, if applicable, accept commands requesting subsequent slots. If the card has zero slots to report (*i.e.,* there are no currencies stored in the card), a reply with status "Finished" is sent (transition from location L2 to location L3) and the feature has completed its processing. If the card has one or more currencies to report, the transition from location L2 to location L4 is taken, returning the balance of one slot to the requester and marking that slot as having been reported.

At location L4 the application waits for the next balance request command, returning to location L2 when it is received. Receiving a "FirstSlot" command at this point will re-initialize the iteration (transition L4

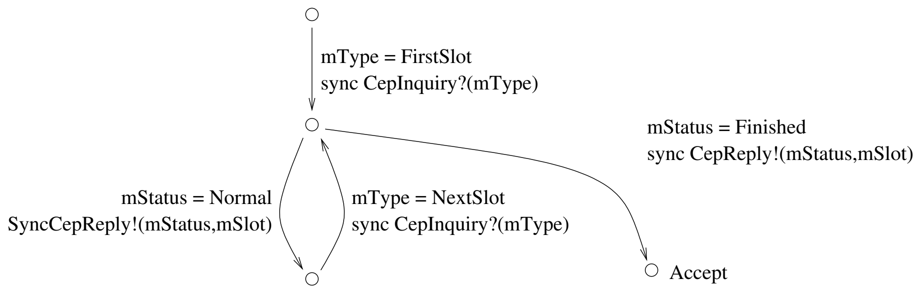

Fig. 3. Test Purpose

to L1). If a command other than "FirstSlot" or "NextSlot" is received a sequence error is generated (transition L4 to L5 to L6).

The test purpose. A test purpose is used to select the behaviors from the specification that are to be exercised by the derived test. Figure 3 illustrates a test purpose that selects from the CEPS slot inquiry feature a test that exercises a single iteration through all the slots of the card, uninterrupted by sequence errors or re-starting of the iteration.

The test purpose is also an IOSTS, with locations and transitions. The generation of tests takes place through the computation of a product between the specification IOSTS and the test purpose IOSTS. Thus, locations in the test are pairs made up of a location from the specification and a location from the test purpose, and transitions between these locations are added when (1) a specification transition action has the same label as a test purpose action (resulting in a guard that is the intersection of the specification and test purpose transition guards), or (2) the specification is capable of advancing on an internal action (with the guard taken directly from the specification edge). The location "Accept" in the test purpose indicates locations in the test IOSTS that should be interpreted as final, indicating successful execution of the test.

The test purpose of Figure 3 was constructed to select behaviors that (1) begin with a "FirstSlot" command, (2) loop as many times as necessary to process all slots, and (3) accept behaviors that conclude with a "Finished" reply. When there are no slots loaded with currency in the purse "Finished" may correctly be generated without any "Normal" replies being generated.

The test. Finally, Figure 4 shows the IOSTS that results from symbolic test generation using the specification of Figure 2 and test purpose of

Figure 3. Note that this test is specific to this test purpose. Different test purposes will generate different tests.

The computation steps carried out are identical to those given in the specification. Actions have had their orientation (*i.e.,* input vs. output) reversed so that the test becomes a generator of commands and a receiver of responses, complementary to an implementation of the specifications.

The location labelled "Pass" in Figure 4 indicates that a correct interaction between the tester and the system under test took place. The symbolic test generation method also generates transitions from every location to a new location "Fail" that absorb incorrect responses from the system under test and lead to the "Fail" state, indicating non-conformance of the implementation. For the sake of clarity of presentation, only one of these transitions is shown in detail in the figure. In fact each possible erroneous input action the tester could receive generates a transition to "Fail" from each location of the graph.

Note that the test shown in Figure 4, like all tests generated by this method, incorporates its own oracle. All of the computation steps necessary to verify the correctness of numeric results are extracted from the specification and used by the tester to verify arguments as they are received. This is in contrast to test generation techniques that simply produce a sequence of inputs to drive the implementation through a specific path.

4 Test Automation

A goal of our work is to produce a fully automated process for symbolic test generation and execution, from high-level specifications down to executable test programs. Our present work is focused on test generation from IOSTS (a formal model intended as an intermediate representation) down to C++ test objects. In this section we present a summary of tool support for test generation, and outline the test execution phase

Tool Support. The test derivation technique described in Section 3 is implemented in a tool called "Symbolic Test Generator," or simply STG. This tool currently implements the functions outlined by a dashed box in Figure 1. This includes (1) reading and parsing IOSTS specifications expressed in a dialect of IF [BFG+99], (2) computing an IOSTS test case from an IOSTS specification and IOSTS test purpose according to the theory of [RdBJ00], and (3) translation of the IOSTS test case to C++,

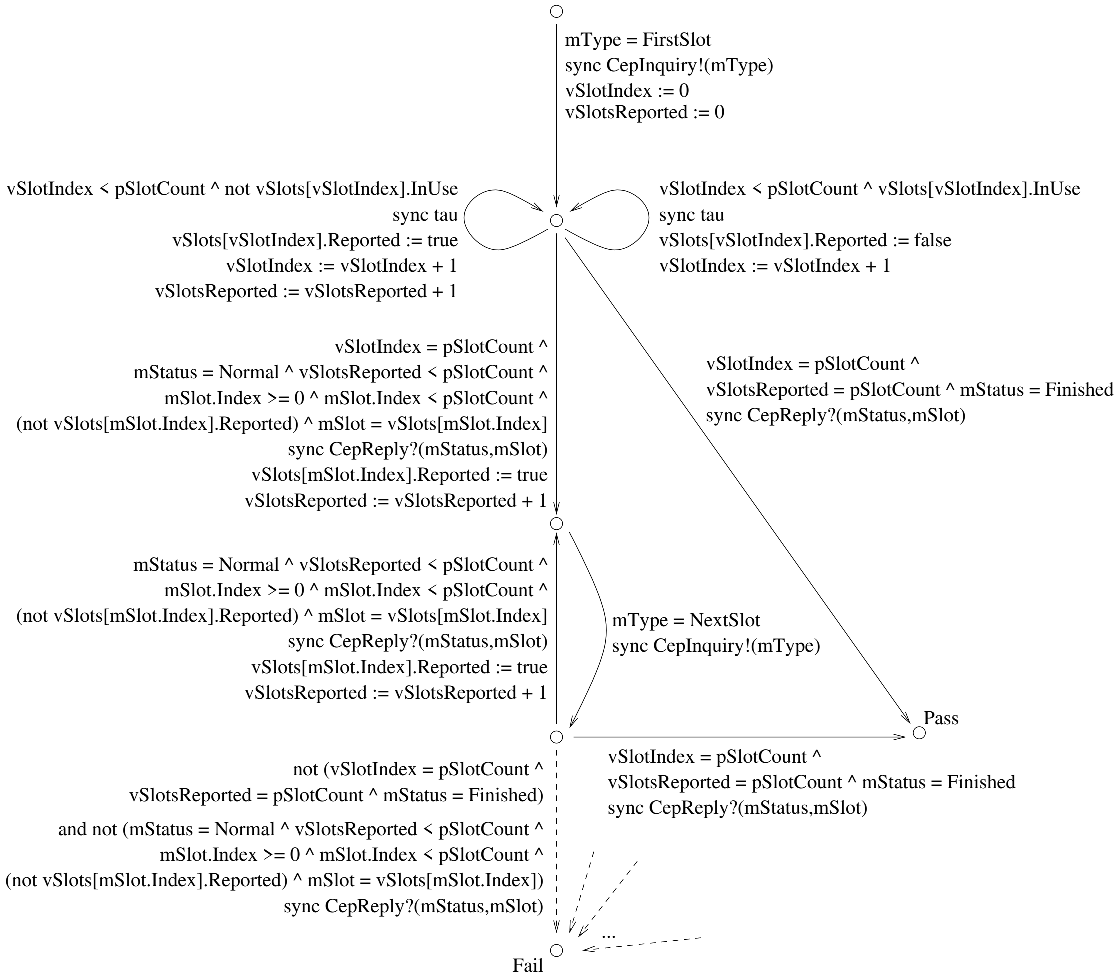
mType = FirstSlot
sync CepInquiry!(mType)
vSlotIndex := 0
vSlotsReported := 0
vSlotIndex < pSlotCount ^ not vSlots[vSlotIndex].InUse
sync tau
vSlots[vSlotIndex].Reported := true
vSlotIndex := vSlotIndex + 1
vSlotsReported := vSlotsReported + 1
vSlotIndex < pSlotCount ^ vSlots[vSlotIndex].InUse
sync tau
vSlots[vSlotIndex].Reported := false
vSlotIndex := vSlotIndex + 1
vSlotIndex = pSlotCount ^
mStatus = Normal ^ vSlotsReported < pSlotCount ^
mSlot.Index >= 0 ^ mSlot.Index < pSlotCount ^
(not vSlots[mSlot.Index].Reported) ^ mSlot = vSlots[mSlot.Index]
sync CepReply?(mStatus,mSlot)
vSlots[mSlot.Index].Reported := true
vSlotsReported := vSlotsReported + 1
vSlotIndex = pSlotCount ^
vSlotsReported = pSlotCount ^ mStatus = Finished
sync CepReply?(mStatus,mSlot)
mStatus = Normal ^ vSlotsReported < pSlotCount ^
mSlot.Index >= 0 ^ mSlot.Index < pSlotCount ^
(not vSlots[mSlot.Index].Reported) ^ mSlot = vSlots[mSlot.Index]
sync CepReply?(mStatus,mSlot)
vSlots[mSlot.Index].Reported := true
vSlotsReported := vSlotsReported + 1
mType = NextSlot
sync CepInquiry!(mType)
Pass
vSlotIndex = pSlotCount ^
vSlotsReported = pSlotCount ^ mStatus = Finished
sync CepReply?(mStatus,mSlot)
not (vSlotIndex = pSlotCount ^
vSlotsReported = pSlotCount ^ mStatus = Finished)
and not (mStatus = Normal ^ vSlotsReported < pSlotCount ^
mSlot.Index >= 0 ^ mSlot.Index < pSlotCount ^
(not vSlots[mSlot.Index].Reported) ^ mSlot = vSlots[mSlot.Index])
sync CepReply?(mStatus,mSlot)
...
Fail

Fig. 4. Test

ready to be compiled, linked to an interface-compatible implementation, and executed.

The inputs of the tool, (1) the IOSTS specification and (2) the IOSTS test purpose(s), are constructed manually by the system engineer using a text editor. Ultimately our goal is to create these inputs automatically based on a pre-existing formal high-level specification model, and higher level descriptions of test objectives such as coverage criteria.

For translation of test cases to executable code the guard expressions of system inputs are limited to formulas in Presburger arithmetic. This is because we depend on the Omega [KMP+] library (for analyzing Presburger formulas) to evaluate the constraints and select specific values for outputs of the tester. In practice this means that expressions on guards of system inputs in the IOSTS specification cannot contain any operations more complex than arithmetic (*I.e.,* addition, subtraction, relational operators, boolean operators) or multiplication by constant factors.

Test Execution The C++ tests are generated based on various assumptions about how the IOSTS model maps into the C++ programming language. For example, inputs and outputs of the IOSTS become function calls and return values, respectively, in the generated C++ code. The system under test is assumed to be a C++ class, and inputs to the system under test are implemented as synchronous method invocations with messages passed through function arguments. The reply of the system under test is the received return value of the method invocation.

Implementations that present an interface corresponding exactly to this model can be compiled with the resulting test program to produce an executable test. Other systems will require the creation of a simple intermediate module that implements the interface described and translates communication into the model expected by the system under test. For example, in the smart card domain, testing of an application resident on a card would require the creation of an interface module to translate method invocations and their parameters into a command APDU, send the APDU to the card over the terminal-to-card interface, wait for the reply APDU, interpret the reply APDU and return the reply value(s) to the method's caller.

Once a test and interface-compatible system are compiled together into executable code, the actual execution of the test consists of (1) supplying any parameter values that are required by the model, and (2) observing the final verdict of the test: "pass," "inconclusive" or "fail."

5 Summary

This paper has presented a symbolic test generation technique which has been successfully applied to test generation for smart card applications. The symbolic generation technique (1) automatically derives test cases in order to check conformance of an implementation with respect to the behavior of a specification selected by the test purposes; (2) automatically determines whether the results of the test execution are correct with respect to the specification. It performs test derivation as a symbolic process, up to and including the generation of test program source code. The reason to use symbolic techniques instead of enumerative is that symbolic test generation allows us to produce (1) more general test cases with parameters and variables which should be instantiated only before the test cases execution, and (2) test cases that are more readable by humans. We presented an example based on the application of these techniques to a feature of the CEPS e-purse specifications.

We are currently using the IOSTS language and STG to develop a large-scale case study based on the CEPS specifications. The IOSTS graphs presented in Section 3 are a small part of the current model, which will ultimately include all of the command processing functions of the CEPS specifications. During the development of this model, STG is being used to incrementally test a prototype implementation of the specifications.

Other directions for our future work include, firstly, the use a higher-level language (*e.g.*, LOTOS, SDL, etc.) for specifying systems at the user level. Second, we plan to work on the implementation of a mechanism to automatically compute test purposes from the system specification using, for example, coverage criteria instead of test purposes written by hand. Third, we continue to work on simplification of test cases, as we want to reduce the size of test cases, which would result in reduced execution time, improved readability, elimination of paths leading to “Inconclusive” verdicts, etc. There are two directions to solve the problem: (1) the test cases could be simplified using automated static analysis and proof strategies; or (2) the test cases could be derived from an abstraction of the system specification instead of the concrete specification.

References

[BFdV+99] A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, and S. Mauw. Formal test automation: a simple experiment. In *International*

Workshop on the Testing of Communication Systems (IWTCS'99), pages 179–196, 1999.

[BFG+99] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. If: An intermediate representation and validation environment for timed asynchronous systems. In *Proceedings of World Conference on Formal Methods, FM'99*, volume 1708 of *LNCS*, pages 307–327, Toulouse, France, September 1999. Springer-Verlag.

[CEP00] CEPSCO. Common Electronic Purse Specifications, Technical Specification (`http://www.cepsco.org`), May 2000.

[ISO88] ISO/IEC. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. Technical Report 8807, International Organization for Standards - Information Processing Systems - Open Systems Interconnection, 1988.

[ISO92] ISO/IEC. International Standard 9646-1/2/3, OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework, 1992.

[IT94] ITU-T. Recommendation Z-100. Specification and Description Language (SDL), 1994.

[JM99] T. Jéron and P. Morel. Test generation derived from model-checking. In *Computer Aided Verification (CAV '99)*, volume 1633 of *LNCS*, pages 108–122, 1999.

[KJG99] A. Kerbrat, T. Jéron, and R. Groz. Automated test genration from S-DL specifications. In *Proceedings of SDL Forum*. Elsevier Science (North Holland), 1999.

[KMP+] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpiesman, and D. Wonnacott. The Omega library interface guide. Available at `http://www.cs.umd.edu/projects/omega`.

[MdB00] H. Martin and L. du Bousquet. Automatic test generation for java-card applets. In *Proceedings of the Java-Card Workshop*, Cannes, September 2000.

[RdBJ00] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357, Dagstuhl, Germany, November 2000. Springer-Verlag.

[RW85] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Tre94] J. Tretmans. A formal approach to conformance testing. In *The 6th International Workshop on Protocol Test Systems*, number C-19 in IFIP Transactions, pages 257–276, 1994.