STG: A Symbolic Test Generation Tool

Duncan Clarke[1], Thierry Jéron[2], Vlad Rusu[2], and Elena Zinovieva[2]

[1] University of South Carolina,
Columbia, South Carolina, USA
[2] IRISA/INRIA Rennes,
Campus de Beaulieu, Rennes, France
dclarke@cse.sc.edu, {jeron|rusu|lenaz}@irisa.fr

Abstract. We report on a tool we have developed that implements conformance testing techniques to automatically derive symbolic tests cases from formal operational specifications. We demonstrate the application of the techniques and tools on a simple example and present case studies for the CEPS (Common Electronic Purse Specification) and for the file system of the 3GPP (Third Generation Partnership Project) card.

1 Introduction

The work that we present is an attempt to leverage the ideas underlying protocol conformance testing [9] and high-efficiency test generation as embodied in the TGV [6] and TorX [2] tools, to automate the generation of tests for smart-card applications. Most existing test generation tools perform their analysis by enumerating the specification's state space. This leads to two problems: (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. To avoid these problems we introduce symbolic generation techniques.

2 STG: The Symbolic Test Generation Tool

Based on the theory of symbolic test generation presented in [8] we have created the STG tool that implements the process illustrated in Fig.1. The system at the user level is described in NTIF a high-level, LOTOS-like language developed by the VASY team, INRIA Rhône-Alpes. The STG tool for symbolic test generation uses IOSTS (Input Output Symbolic Transition Systems) [8] as an internal model for reactive systems. To obtain such a model, the system written in NTIF is automatically translated into IOSTS (*cf.* Fig.1 above the dashed box).

Currently the STG tool supports two processes (*cf.* Fig.1), which are briefly described below.

Symbolic test generation. The process of symbolic test generation takes a specification of the system together with a test purpose and produces a symbolic test

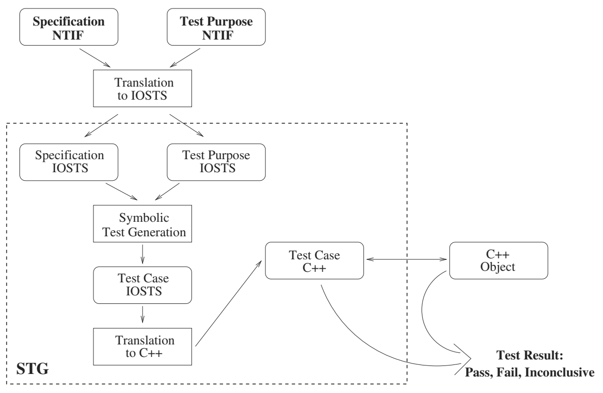

Fig. 1. Symbolic Test Generation Process

case, which is a reactive program covering all the behaviors of the specification that are targeted by the test purpose. A detailed description of symbolic test generation and its properties can be found in [8].

IOSTS to C++. To obtain an executable test, the abstract, symbolic test case obtained after symbolic test generation is translated into a concrete test program capable of interacting with an implementation interface-compatible with the original specification. The test program is then ready to be compiled and linked with the implementation for test execution. The results of a test execution are “Pass”, which means no errors were detected and the test purpose was satisfied, “Inconclusive” - no errors were detected but the test purpose was not satisfied, or “Fail” - an error was detected. Here, “error” means a non-conformance between the implementation and the specification. The conformance relation is defined in [9,8]

We illustrate the symbolic test generation process on a simple example.

The specification. Fig.2 presents the IOSTS specification of a coffee machine. As shown in the figure, the IOSTS is made up of control states called *locations* and of transitions between locations that describe either input, output, or internal actions and manipulate symbolic data. A transition can be fired if its *guard* is true, then executes its action and performs assignments that set its variables to new values

The machine starts in the *Begin* location with the initial condition $pPrice > 0$, that is, the price of any beverage dispensed by the machine is strictly positive. Then, the machine moves to the *Idle* location by initializing the *vPaid* variable, which memorizes the amount already paid. Next, the machine expects a coin, denoted by the *Coin?* input action that carries in *mCoinValue* the value of the inserted coin, and the variable *vPaid* is increased by *mCoinValue* and the machine moves to the *Pay* location. If the payment is not enough

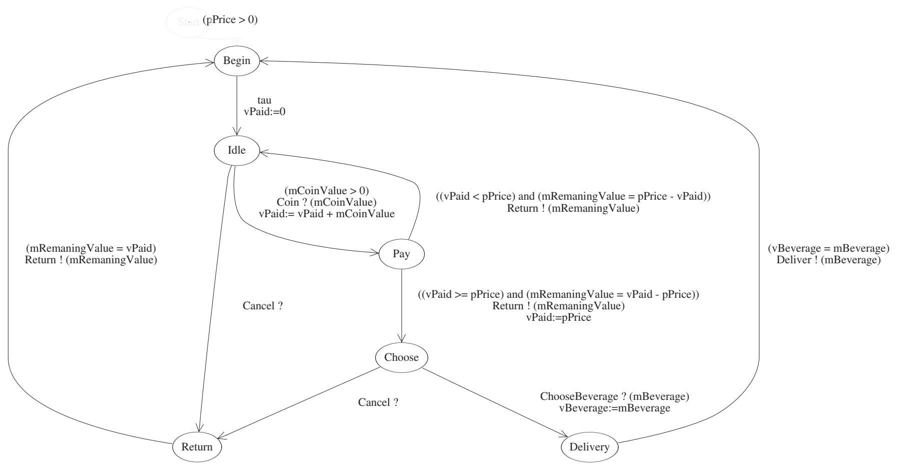
Start
(pPrice > 0)
Begin
tau
vPaid:=0
Idle
(mCoinValue > 0)
Coin ? (mCoinValue)
vPaid:= vPaid + mCoinValue
((vPaid < pPrice) and (mRemaningValue = pPrice - vPaid))
Return ! (mRemaningValue)
Pay
(mRemaningValue = vPaid)
Return ! (mRemaningValue)
(vBeverage = mBeverage)
Deliver ! (mBeverage)
Cancel ?
((vPaid >= pPrice) and (mRemaningValue = vPaid - pPrice))
Return ! (mRemaningValue)
vPaid:=pPrice
Choose
Cancel ?
ChooseBeverage ? (mBeverage)
vBeverage:=mBeverage
Return
Delivery

Fig. 2. Example of IOSTS: a Coffee Machine

i.e., $vPaid < pPrice$, the machine moves back to the *Idle* location and returns (through the *Return!(mRemaningValue)* output action) the difference between the paid amount and the cost of beverage, *i.e.* $pPrice - vPaid$. Otherwise, the machine moves to the *Choose* location and returns in *mRemaningValue* the difference between *vPaid* and *pPrice*. In the *Choose* location, the machine waits for the choice of the beverage (tea or coffee), then delivers the beverage, and moves back to the *Begin* location. Note that in locations *Idle* and *Choose*, the *Cancel* button can be pressed, in which case the machine returns the amount already paid and moves back to the initial location.

The test purpose. Fig.3 presents one possible test purpose for the coffee machine, which describes behaviors where the machine delivers coffee and the user does not introduce coins more than once and does not cancel. An accepted behavior is indicated by arrival at location *Accept*. The test purpose rejects behaviors that correspond to delivery of tea to the user, or pressing the *Cancel* button, or inserting more than one coin. Note that rejected behaviors are not necessarily erroneous, they are just behaviors that are not targeted by the test purpose.

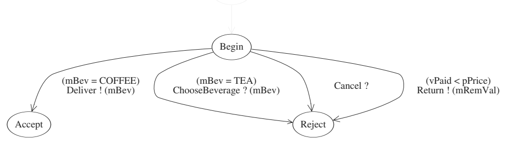

Fig. 3. Test Purpose

The test case. Fig.4 presents the IOSTS test case automatically generated by STG which covers all the behaviors of the specification (*cf.* Fig.2) that are targeted by the test purpose of (*cf.* Fig.3). Note that the test case is limited to the behaviors targeted by the test purpose: it accepts only one payment and does not exercise pressing the cancel button or require delivery of tea (*cf.* Fig.4).

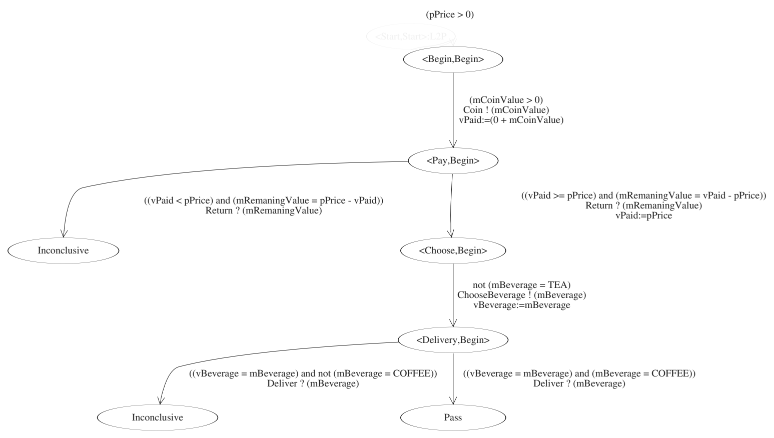
(pPrice > 0)
<Start,Start>:L2P
<Begin,Begin>
(mCoinValue > 0)
Coin ! (mCoinValue)
vPaid:=(0 + mCoinValue)
<Pay,Begin>
((vPaid < pPrice) and (mRemaningValue = pPrice - vPaid))
Return ? (mRemaningValue)
((vPaid >= pPrice) and (mRemaningValue = vPaid - pPrice))
Return ? (mRemaningValue)
vPaid:=pPrice
Inconclusive
<Choose,Begin>
not (mBeverage = TEA)
ChooseBeverage ! (mBeverage)
vBeverage:=mBeverage
<Delivery,Begin>
((vBeverage = mBeverage) and not (mBeverage = COFFEE))
Deliver ? (mBeverage)
((vBeverage = mBeverage) and (mBeverage = COFFEE))
Deliver ? (mBeverage)
Inconclusive
Pass

Fig. 4. Test Case

Test case execution. The test case is now ready to be translated to C++ and to be linked and executed on an implementation under test. For example, suppose the implementation has a method as shown in Fig.5, which corresponds to the *ChooseBeverage!* output in the test case (*cf.* Fig.4). The returning value of this function corresponds to the *Deliver?* input. Then, the delivery of tea after the coffee request (*cf.* Fig.5: lines 4, 5) denotes such a non-conformance [9,8], *i.e.* a difference between the implementation and the specification. As a consequence, by execution the test case on such an implementation we get a “Fail” verdict.

STG uses OMEGA [7] to detect unsatisfiable guards for simplifying the test cases, and *dotty* [5], a tool for drawing graphs to view IOSTS in graphical form. Figures 2, 3, 4 were produced by *dotty*.

3 Case Studies

The STG tool was applied for testing simple versions of the CEPS (Common Electronic Purse Specification) [3] and of the file system of the 3GPP (Third Generation Partnership Project) card [1].

```
...
BeverageType ChooseBeverage(BeverageType mBeverage){
 cerr << "ChooseBeverage(";
 if(mBeverage == COFFEE){
  cerr << "TEA)";
  return TEA;
 }
 if(mBeverage == TEA){
  cerr << "TEA)";
  return TEA;
 }
}
...
```

Fig. 5. Implementation of the "ChooseBeverage" Function

The CEPS is a standard for creating inter-operable multi-currency smart card e-purse systems. The specification of the CEPS, which is presented as an IOSTS model, has about 100 transitions and 40 variables of various types, including structured types built with records and arrays. The feature that we generate tests for is the "CEP Inquiry - Slot Information" specified in Section 8.7.1 of the CEPS technical specifications [3]. It provides a means for iterating through the slots, where each slot corresponds to one currency and its respective balance. The paper [4] presents our results of this experiment.

The 3GPP card is a multi-applications microprocessor smart card. We generate tests for the file system of the card, which is organized as follows: it has one *master* file (a root of the system) which contains *dedicated* (directory files), *application dedicated* (special directory files for the applications), and different kinds of *elementary* files where data are organized either as a sequence of bytes or a set of records. The current specification of the file system for the 3GPP card allows to create files on the card, to search a record in the files, and to get a response from the card after the search is performed. The specification has been written in the NTIF language, and automatically translated into the IOSTS model, which has about 100 transitions, 50 locations, and 30 variables of various types, including structured types built with records and arrays.

Using STG we automatically generate executable test cases for these systems. The test cases are executed on implementations of the systems, including mutants. Various errors in the source code of the mutants were detected.

4 Summary

This paper has presented a tool that automates the derivation of test cases in order to check conformance of an implementation with respect to the behaviors of a specification targeted by test purposes; and 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 produces (1) more general test cases with parameters and variables which need to be instantiated only at the test execution time, and (2) test cases that are more readable by humans.

We have presented a simple example that demonstrates the application of the method and the tool to a software testing problem, and reported case studies for the CEPS and for the file system of the 3GPP card.

References

1. 3GPP. Third Generation Partnership Project (`http://www.3gpp.org`).
2. 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.
3. CEPSCO. Common Electronic Purse Specifications, Technical Specification (`http://www.cepsco.org`), May 2000.
4. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *Proceedings of the International Conference on Research in Smart Cards*, volume 2140 of *LNCS*, pages 58–70, Cannes, France, September 2001.
5. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, September 2000.
6. 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.
7. 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`.
8. V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods*, volume 1945 of *LNCS*, pages 338–357, Dagstuhl, Germany, November 2000. Springer-Verlag.
9. 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.