Automatic generation of symbolic test purposes

Assia Touil* — **Christophe Gaston**** — **Pascale Le Gall***

*Université d'Évry-Val d'Essonne, LaMI CNRS UMR 8042,
523 pl. des Terrasses F-91025 Évry Cedex, France
atouil@lami.univ-evry.fr
legall@lami.univ-evry.fr

**CEA/LIST/SOL Saclay
F-91191 Gif sur Yvette Cedex
christophe.gaston@cea.fr

Abstract. *We present a technique dedicated to compute automatically sets of test purposes from IOSTS specification. A coverage criterion, namely the* inclusion criterion, *is defined. Inclusion criterion is dedicated to decide whether set of test purposes are sufficiently complete or not. Test case generation is then studied.*

1. Introduction

We focus on conformance testing [1] [2] for reactive systems specified with Input Output Symbolic Transition Systems [3] [4] [5]. Such systems are continuously sending and receiving data through channels. The Implementation Under Test (IUT) is tested by observing whether sequences of Input Output exchanged with its environment conform to the specification. Test cases are extracted from the specification. A test case is a structuring of inputs and intended outputs. When the test case is executed, a verdict is emitted to express whether the implementation behaved as intended in the test case definition or not. Roughly speaking, test cases extraction process can be parameterized by two kinds of *stopping criteria*. The first kind is a static one: the idea is to ensure that a set of selected test cases is such that all the specification has been "covered". This coverage notion is characterised by a so called *coverage criteria* [6] which allows to quantify on some syntactical elements of the specification to be covered by test cases (*e.g.* all transitions, all paths of a certain length given as parameter. . .): thus the process stops when the specification is "covered" according to the coverage criteria. The second kind of stopping criteria is functional: the idea is to characterize a set of functional properties, denoting typical intended behaviours of the IUT and for each of these intended behaviours to construct a test case which is supposed to activate the behaviour: thus the process stops when at least one test case is associated to each intended behaviours. Classically such functional properties are called *test purposes* [3]. Test purpose characterization are generally generally an expert matter. In this contribution, we propose to automatically build test purposes and derive test cases by analysing specifications with symbolic execution techniques. Indeed, symbolic execution allows us to extract abstract behaviours. These behaviours are good candidates in order to become test purposes. For that, we use the AGATHA tool [7].

2. A symbolic framework for communicating systems

2.1. Input Output Symbolic Transition Systems

IOSTS (Input Output Symbolic Transition Systems)[4] allow the specifier to describe communicating systems with data by means of automata. Let us consider an ATM system built over the two communicating automata depicted in Figure 1. The IOSTS on the left side (`UserInterface`) is the specification of the interface with the bank user and the IOSTS of the right side (`InternalSystem`) specifies treatments made by the system to allow or forbid cash withdrawal.

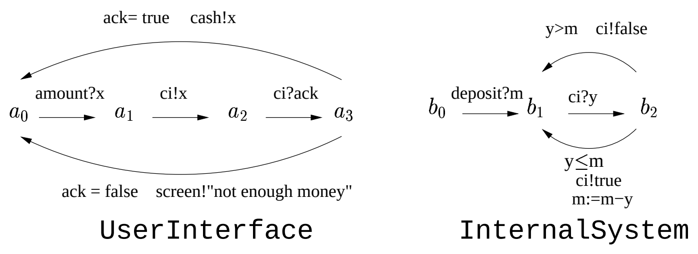

Figure 1. an ATM system

An IOSTS is composed of a set of states, an initial state, a variable set and a transition set.

- In Figure 1, $\{a_0, a_1, a_2, a_3\}$ (resp. $\{b_0, b_1, b_2\}$) is the set of states of the `UserInterface` (resp. `InternalSystem`) and a_0 (resp. b_0) is its initial state.
- In the `UserInterface`, variables are ack (a boolean variable which is true when the system allows withdrawal and false otherwise) and x (corresponding to the amount asked by the user). In the `InternalSystem`, variables are m (for the available amount of money in the user bank account) and y (dedicated to store the amount asked by the user).
- Transitions are made of a *guard*, a *communication term* and an *assignment*, each of them possibly being empty:
  - *Guards* are formulas on variables of the IOSTS. A guard conditions whether the associated transition can be executed or not. For example, in the `InternalSystem`, the transition from b_2 to b_1 in the upper part of Figure 1 can be executed only if the condition $y > m$ is satisfied.
  - *Communication terms* denote value exchanges. For example, in the `InternalSystem`, the transition from b_1 to b_2 contains the communication term $ci?y$ modelling a reception: a value is received through the channel ci and stored on the variable y. In the `UserInterface`, the transition from a_1 to a_2 contains the communication term $ci!x$ modelling an emission: the value assigned to x is sent through the channel ci.
  - *Assignments* are performed when the transition is executed. For example, in the `InternalSystem`, in the transition from b_2 to b_1 in the lower part of Figure 1, $m := m - y$ means that the value $m - y$ is assigned to the variable m when the transition is executed.

The ATM system is the composition of the two IOSTS `UserInterface` and `InternalSystem`. The IOSTS corresponding to the whole system is built using "handshake" communications by synchronising emissions and receptions on common channels (also called *internal channels*). ci is the unique internal channel to be considered while the channels $amount$, $cash$, $screen$ and $deposit$ are *external channels* representing communications with the environment (normally composed by the user and the bank counter). States of the ATM system are made of an `UserInterface` state and of an `InternalSystem` state (for example, the composite state a_0b_1). The ATM initial state, a_0b_0, is made of the two initial states. The variable set is the union of the two variable sets. The ATM transitions are either internal transitions, denoted by the internal communication term τ, resulting from the synchronisation of an emission and a reception on the same internal channel (for example $a_1b_1 \xrightarrow{\tau} a_2b_2$), or (normal) transitions inherited from a transition of one of the two subsystems while the second subsystem is stationary (for example, $a_0b_0 \xrightarrow{amount?x} a_1b_0$).

When one wants to study systems composed of communicating IOSTS (for example to extract test cases), it may be impracticable to construct the resulting product due to its intrinsic large size. We propose to use symbolic execution based techniques to provide a corresponding compact IOSTS representation.

2.2. Symbolic execution

Our approach provides the specifier with a set of structured behaviours deduced from the system. These behaviours are given under the form of sequences of consecutive transitions starting from the initial state. Constraints, called *path conditions*, are associated to each behaviour.

Figure 2 gives the set of all computed behaviours associated to the system presented in figure 1.

$$\begin{tikzcd}
 & init : (a_0b_0, true, \{x \to x_0\ ack \to ack_0\ y \to y_0\ m \to m_0\}) \arrow[dl, "\text{deposit?m}"'] \arrow[dr, "\text{amount?x}"] & \\
A_0 : (a_0b_1, true, \{x \to x_0\ ack \to ack_0\ y \to y_0\ m \to m_1\}) \arrow[dr, "\text{amount?x}"'] & & A'_0 : (a_1b_0, true, \{x \to x_1\ ack \to ack_1\ y \to y_0\ m \to m_0\}) \arrow[dl, "\text{deposit?m}"] \\
 & A_1 : (a_1b_1, true, \{x \to x_1\ ack \to ack_1\ y \to y_0\ m \to m_1\}) \arrow[d, "\tau"] & \\
 & A_2 : (a_2b_2, true, \{x \to x_1\ ack \to ack_1\ y \to x_1\ m \to m_1\}) \arrow[dl, "\tau"'] \arrow[dr, "\tau"] & \\
A_3 : (a_3b_1, x_1 \leq m_1, \{x \to x_1\ ack \to true\ y \to x_1\ m \to m_1 - x_1\}) \arrow[d, "\text{ack = true}\ \text{cash!x}"] & & A_5 : (a_3b_1, x_1 > m_1, \{x \to x_1\ ack \to false\ y \to x_1\ m \to m_1\}) \arrow[d, "\text{ack=false}\ \text{screen!\"not enough money\"}"] \\
A_4 : (a_0b_1, x_1 \leq m_1, \{x \to x_1\ ack \to true\ y \to x_1\ m \to m_1 - x_1\}) & & A_6 : (a_0b_1, x_1 > m_1, \{x \to x_1\ ack \to false\ y \to x_1\ m \to m_1\})
\end{tikzcd}$$

Figure 2. Symbolic execution

At the initialisation step, the system is in the state a_0b_0. The initial condition is the $true$ condition since at this step, there are no constraints on the variables. Each variable handled in the ATM system is assigned by a constant symbol: x by the constant x_0 (denoted by $x \to x_0$), ack by the constant $ack_0, \ldots$ All the constant symbols introduced in the initial state are conventionally indexed by 0 and called initialisation constant symbols. This information is stored in a so-called *symbolic state* $init$ at the root of the tree. Note that the second component state $init$ is a formula: $true$. In any symbolic state this second component denotes the path condition: that is the constraint on the constant symbols to reach the symbolic state. In the case of $init$ there are no constraints, thus the path condition is $true$.

Now, the `UserInterface` (resp. `InternalSystem`) subsystem can evolve if it receives a value for the variable x (resp. m) from the environment through the channel $amount$ (resp. $deposit$). It corresponds to a symbolic transition yielding to a new symbolic state, denoted by A'_0, made of the composite state a_1b_0, the path condition $true$ and the variable assignment is modified by the new assignment $x \rightarrow x_1$ where x_1 is a new constant symbol denoting the value sent by the environment. The symbolic state A_0 obviously corresponds to the reception $deposit?m$ of the `InternalSystem` subsystem.

A_1 is the next symbolic state when the two independent reception transitions $deposit?m$ and $amount?x$ have been executed. Independence of transitions means that they can be executed in an arbitrary order while leading to the same state (observable interleaving). This is the case since the two automata share no variables. Let us note that the path condition remains "$true$" since there are no guards on both intermediate transitions.

The next symbolic transition is an internal communication τ, obtained by synchronizing the two subsystems on the channel ci. This "handshake" communication is executed by generating the node A_2 made of the composite state a_2b_2, the path condition $true$ and the variable assignment modified by the new assignment $y \rightarrow x_1$, x_1 being the current symbolic value of the variable x, sending by the `UserInterface` to the `InternalSystem`. The symbolic transitions issued from A_2 are still internal communications on the same channel ci, but corresponding to a sending of the `InternalSystem`. According to the different symbolic values of the variables y and m, the sent value may be true or false. The current assignment of the variables y and m ensures the condition $y \leq m$ if and only if the associated symbolic values, x_1 for the variable y and m_1 for the variable m satisfy $x_1 \leq m_1$. This last condition becomes the path condition for the symbolic transition from A_2 to A_3 which is made of the composite state a_3b_1 and the new assignment $ack \rightarrow true$ induced by the "handshake" communication. The symbolic state A_5 is built in a similar way for the case in which $y \leq m$ does not hold.

Symbolic execution is pursued in Figure 2 by creating two new symbolic states A_4 et A_6 from the two previous states A_3 and A_5. The ATM system can be unfolded in an infinite execution tree. So, symbolic execution techniques are helpful to provide a set of symbolic behaviours, which are representative of all concrete behaviours for which system variables are assigned by concrete values (in the set of natural numbers for example). However, to be fully exploitable, one should be able to cut the computation of symbolic behaviours. A first natural way is to eliminate each branch issued from a symbolic state provided with an unsatisfiable path condition. A second way is to recognise redundant behaviours. This is done using the so called *state inclusion* criterion.

Let us consider the symbolic node A_4.

- We remark that this symbolic state is made of the composite state a_0b_1 as in the symbolic state A_0.

- Moreover, the associated path condition ($x_1 \leq m_1$) and the variable assignment ($x \to x_1 \ldots m \to m_1 - x_1$) allows us to characterise a set of constraints on the variables x, ack, y and m. These constraints are $m \geq 0$, $ack = true$ and $x = y$. In the same way, we remark that there are no constraints on these variables in the symbolic state $init$.

In this situation, one says that *the symbolic state A_4 is included in A_0*. This precisely defines *the inclusion criterion between symbolic states*. When a symbolic state A is detected as included in another symbolic state B, then A is removed from the construction and all transitions leading to A are replaced by transitions leading to B. In the case of the ATM system, the states A_4 and A_6 are then both replaced by the state A_0. By applying such a reduction mechanism, we lost some information concerning path condition or variable assignment but we preserve all the possible "future" execution paths (for example, any execution computed from A_4 or A_6 is an execution that can be executed from A_0).

In Figure 3 we can see, the IOSTS representing the reduction of the symbolic execution of Figure 2. Symbolic states have been replaced by their names ($init$, $A_0, \ldots$). States A_4 and A_6 have been replaced using the inclusion criterion and the transitions targeted on these two states are dotted. This IOSTS has the same symbolic execution tree than the original ATM system.

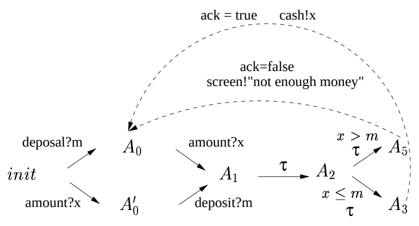

Figure 3. Automata of Symbolic execution

The AGATHA tool ([8]) allows to compute for each IOSTS a corresponding IOSTS which is reduced using elimination of unsatisfiable path conditions and inclusion criterion on symbolic states. This is done thanks to mechanisms using convex polyhedra in the frame of Presburger arithmetic. Note however, that the methodology presented in this paper could be theoretically applied for any decidable data theory. Each redundant symbolic state to be replaced by inclusion criterion is replaced by the first symbolic state in which it is included in, according to the coverage strategy (depth or width).

Moreover, the AGATHA tool is capable of detecting deadlock states and symbolic states with non determinism choices for the transitions to be fired. It also analyses IOSTS with respect to reachability issues. Using constraints solving techniques, AGATHA can compute a numerical instance of any selected branch of the symbolic execution tree.

3. Testing conformance for IOSTS

3.1. Test purposes

Conformance testing [1] is based on a conformance relation between the IOSTS modelling the implementation under test (IUT) and the IOSTS denoting the specification (SP). The most used conformance relation is the so-called ioco relation, which requires that at any time, the IUT outputs are among the outputs allowed by SP, the absence of outputs (quiescence) being denoted by a special output symbol, often denoted by δ. A test case is a IOSTS interacting with the IUT and giving some test verdicts according to some test purpose: $pass$ if the test purpose is recognised, $fail$ if the test purpose is rejected and $inconc$ (for inconclusive) if nothing can be said with respect to the considered test purpose. A test purpose allows the tester to select a property to be tested. Given a test purpose under the form of a tree with leaves marked with $accept$ or $reject$ for test target, the goal consists in building a test case in accordance with both the specification and the test purpose. In most works [3], test purposes are supposed to be given by an expert. Thus, tested behaviours may be "clever" but the price to pay is that the specification is prone to be covered partially. On the contrary, test purposes proposed in our approach are computed automatically, ensuring a coverage of the specification (thanks to symbolic states inclusion mechanism).

By applying symbolic execution techniques and succeeding in computing a finite reduced symbolic tree, then we get for free a set of finite symbolic paths which can be naturally selected to build test purposes in the sense of [3]. For that, it suffices to replace each symbolic transition whose target state is a state previously encountered by a $accept$ state. To illustrate, let us consider the IOSTS in Figure 3 representing the reduction of the symbolic execution tree. If we replace the two dotted transitions $A_5 \to A_0$ and $A_3 \to A_0$ by respectively the two transition $A_5 \to accept$ and $A_3 \to accept$, then we naturally get a test purpose dedicated to test symbolic properties of the specification (see Figure 4).

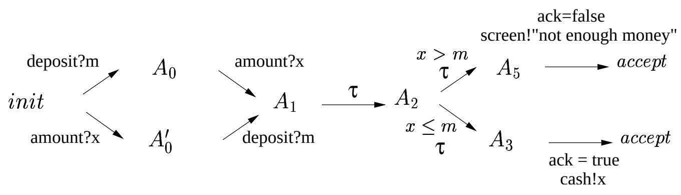

Figure 4. the test purpose

3.2. Test cases

In this section, we investigate the case of deterministic specifications containing controllable variables. Intuitively, a specification is said to be deterministic if for all transitions of same source, guards are mutually exclusive. Variables are controllable if in the reduced execution tree, there are no symbolic states containing path conditions built on initialisation constant symbols (which are in our case indexed by 0). Thus, executions do not depend on internal unobservable values assigned to the system variables. The ATM system is deterministic since the two transitions $A_2 \rightarrow A_3$ and $A_2 \rightarrow A_5$ are exclusive (their respective guards are incompatible $y \leq m$ and $y > m$). All its variables are controllable since all path conditions only depend on values m and x received respectively through the channels *deposit* and $amount$.

Under these hypotheses on the specification, it suffices to select within the test purpose any path from the initial state to an $accept$ state. Let us choose for example the path $init\ A_0\ A_1\ A_2\ A_3\ Accept$. It corresponds to the input output sequence : $deposit?m_1\ amount?x_1 cash!x_1$, the τ transitions being left implicit. At this path, the symbolic execution tree gives the path condition $x_1 \leq m_1$. It suffices to execute the IO sequence $deposit?m_1\ amount?x_1 cash!x_1$ with concrete values m_1 and x_1 satisfying the corresponding path condition $x_1 \leq m_1$. Such values can be generated with the AGATHA tool (using constraints solving techniques). In practice, if m_1^G and x_1^G are such generated concrete values (for example, resp. 100 and 50), it suffices to send the value m_1^G through the channel $deposit$, then to send the value x_1^G through the channel $amount$, then it is expected that the IUT sends the value x_1^G on the channel $cash$. If this is encoded in a test case, this scenario is labelled by $pass$: the IUT passes the IO sequence with success. Any other scenario (other intermediate events or other final emission of the IUT) is labelled by $fail$. In fact, as the specification is deterministic and has only controllable variables, there are no inconclusive scenarios. The expected behaviour of the IUT in response to the actions of the test case is unique.

4. Conclusion

In this paper, we have presented a functional criterion to extract a set of test purposes (namely the inclusion criterion). This kind of criterion can be of great help for experts that usually define test purposes manually. We have shown how to define test cases for each test purpose when dealing with deterministic specifications with initialised variables. This constraints on determinism and initialisation can be relaxed using techniques similar to those presented in[3].

References

[1] J. Tretmans, “Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation,” *Computer Networks and ISDN Systems*, vol. 29, pp. 49–79, 1996.

[2] M. Yannakakis and D. Lee, “Testing finite state machines,” in *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. ACM Press, 1991, pp. 476–485.

[3] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva, “Symbolic test selection based on approximate analysis,” in *11th Int. Conference on Tools and Algorithms for tthe Construction and Analysis of Systems (TACAS)*, Edinburgh, Scottland, April 2005.

[4] V. Rusu, L. du Bousquet, and T. Jéron, “An approach to symbolic test generation,” in *IFM ’00: Proceedings of the Second International Conference on Integrated Formal Methods*. London, UK: Springer-Verlag, 2000, pp. 338–357.

[5] L. Frantzen, J. Tretmans, and T. A. Willemse, “Test generation based on symbolic specifications,” in *FATES 2004*, ser. LNCS, J. Grabowski and B. Nielsen, Eds., no. 3395. Springer-Verlag, 2005, pp. 1–15.

[6] C. Gaston and D. Seifert, “Evaluating coverage based testing,” in *Model-based testing of reactive systems : advanced lectures*, ser. LNCS, J.-P. K. Manfred Broy, Bengt Jonsson and al., Eds., vol. 3472 / 2005. Springer-Verlag, 2005, p. 293.

[7] C. Bigot, A. Faivre, J.-P. Gallois, A. Lapitre, D. Lugato, J.-Y. Pierron, and N. Rapin, “Automatic test generation with agatha.” in *TACAS*, 2003, pp. 591–596.

[8] N. Rapin, C. Gaston, A. Lapitre, and J.-P. Gallois, “Behavioural unfolding of formal specifications based on communicating automata,” in *Proceedings of first Workshop on Automated technology for verification and analysis*, Taiwan, 2003.