Testing a deterministic implementation against a non-controllable non-deterministic stream X-machine

Robert M Hierons[1] and Florentin Ipate[2]

[1]School of Information Systems, Computing, and Mathematics,
Brunel University, Uxbridge, Middlesex, UB8 3PH
[2]Department of Computer Science and Mathematics,
University of Pitesti, Str Targu din Vale 1, 0300 Pitesti, Romania

Abstract. A stream X-machine is a type of extended finite state machine with an associated development approach that consists of building a system from a set of trusted components. One of the great benefits of using stream X-machines for the purpose of specification is the existence of test generation techniques that produce test suites that are guaranteed to determine correctness as long as certain well-defined conditions hold. One of the conditions that is traditionally assumed to hold is controllability: this insists that all paths through the stream X-machine are feasible. This restrictive condition has recently been weakened for testing from a deterministic stream X-machine. This paper shows how controllability can be replaced by a weaker condition when testing a deterministic system against a non-deterministic stream X-machine. This paper therefore develops a new, more general, test generation algorithm for testing from a non-deterministic stream X-machine.

Keywords: Stream X-machine; non-determinism; testing; controllability.

1. Introduction

A formal model or specification can form the basis for automated test generation and this can reduce both the cost of testing and the scope for errors in the testing process. Since many systems have a finite state structure there has been much interest in testing from a finite state machine (see, for example, [DSA$^+$99, Hie03, HU06, LY96, PBG04, PY05a, PY05b, UW03]). However, finite state machines are not always appropriate

Correspondence and offprint requests to: Robert M Hierons, School of Information Systems, and Computing Mathematics, Brunel University, Uxbridge, Middlesex, UB8 3PH, e-mail: rob.hieons@brunel.ac.uk for modelling systems that have an internal memory: extended finite state machines (EFSMs) might instead be used.

The stream X-machine (SXM) is a type of EFSM. It describes a system using a finite set of states, an internal store, called memory, and a number of transitions between the states, labelled by relation names. Various case studies [HI98, KEK03] have demonstrated the value of the stream X-machine as a specification method, especially for interactive systems. Many variants and subclasses of stream X-machines [HI98, ABC$^+$02, BGH03] and communicating Stream X-machine [CGV00] have been defined and investigated. Communicating stream X-machines have been used for modelling biological systems such as ant foraging networks [JHR04] or epithelial cells [SH06] or the dynamic organization of biology-inspired multi-agent systems [SGK04]. More recently, NASA has discussed using a combination of Communicating Stream X-Machines and the process calculus WSCSS in the design and testing of swarm satellite systems [RHRT05].

Associated with SXMs is a software development approach [BHI$^+$06] in which a system is built from trusted components and communication between the components is modelled by the SXM's memory. Testing is black-box: all we can observe are inputs sent to the implementation under tests and outputs it produces. Testing can be seen as a process of checking that the components have been integrated in an appropriate way (see, for example, [BGGV99, BH01, Hol93]). Naturally, the trusted components might have been produced in a previous SXM based development phase and thus SXMs can be used in incremental development. The components may also have been previously tested using some other approach and recent results have shown that the generation of tests for the components of a SXM Z can be integrated into the process of generating a test suite from Z [Ipa04]. Naturally, the reliance on trusted components fits well with the increasing use of component based software development methodologies. A key benefit of this "divide et impera" strategy is that, unlike many other EFSM based test generation methods [LY96], the stream X-machine based approach does not involve the construction of the equivalent finite state machine (whose states are the state/memory pairs of the stream X-machine), thus avoiding the state explosion problem associated with this construction.

A key aspect of the SXM approach is that as long as certain conditions hold, it is possible to produce a finite test suite that determines correctness: if the implementation under test (IUT) is faulty and satisfies these conditions then the test suite *must* lead to a failure. The conditions can be divided into two types: *design for test conditions* that place restrictions on the SXM model/specification from which test are to be generated and *test hypotheses* that place restrictions on the IUT.

One of the traditional design for test conditions is that the SXM is *controllable*: all paths through the SXM model are feasible. While this is an extremely useful property that significantly simplifies test generation, many specifications are not controllable. While additional inputs can be added to a specification in order to ensure that it is controllable, this may be undesirable. There has been recent work on transforming an EFSM to one in which all paths are feasible but this approach relies on the operations and guards being linear [FUDA03, DU04]. It has recently been shown that controllability can be weakened to input-uniformity when considering deterministic SXMs [Ipa06]. Input-uniformity essentially says that each processing relation changes the memory in a uniform manner: if there is a sequence p of processing relations and a processing relation ϕ such that $p\phi$ is feasible then the feasibility of ϕ after p is not affected by the particular choice of input sequence used to trigger p. It has been argued that almost all SXM specifications satisfy this weaker condition [Ipa06].

The initial work on using SXMs in software development only considered deterministic SXMs (see, for example, [HI98]). However, while many implementations are deterministic, non-determinism aids abstraction and is highly appropriate for specifications. There has thus been recent interest in testing from non-deterministic SXMs [HH00, HH04, IH00]. The observable behaviour of a SXM is that of an input/output transducer and so, unlike in the case of finite automata, it is not possible to convert a non-deterministic SXM into a functionally equivalent SXM. For example, if a non-deterministic SXM responds to an input of 1 in its initial state and memory with either output 0 or output 1 then it is not equivalent to a deterministic SXM.

All of the previous work regarding testing from a non-deterministic SXM assumes that the SXM is controllable. The main contribution of this paper is to show how controllability can be weakened to input-uniformity when testing a deterministic IUT against a non-deterministic SXM. We thus weaken the controllability assumption and show that this has significant ramifications for test generation since the SXM may have infeasible paths: a test generation algorithm must avoid such infeasible paths while still achieving the required test objective. As a result, we require a different test generation algorithm and this paper gives such an algorithm and shows that the resultant test suite achieves full fault coverage with respect to the fault domain defined by the test hypotheses: for every IUT that satisfies the test hypotheses, if the IUT passes the test suite then it must conform to the SXM specification.

This paper is structured as follows. Section 2 defines SXMs and introduces terminology and notation used throughout the paper. Section 3 describes the design for test conditions and test hypotheses used in this paper and Section 4 explains how we can find sequences to reach and distinguish states of a SXM. Section 5 describes the product machine, which provides the basis of our test generation algorithm, and Section 6 defines test processes that can be used to determine whether a specified sequence of operations has been implemented. Section 7 then gives the test generation algorithm, Section 8 gives the complexity of the approach, and Section 9 draws conclusions.

2. Preliminaries

2.1. Notation

Given a finite set B, $card(B)$ denotes the number of elements in B. Given a finite alphabet B, B^* denotes the set of all finite sequences of elements of B including the empty sequence ϵ. We denote by B^k the set of sequences of length k of elements of B and $B[k]$ is the set of sequences of elements of B that have length less than or equal to k.

Given two sequences b_1 and b_2, the concatenation of b_1 and b_2 is denoted b_1b_2. Given two sets B_1 and B_2 of sequences, $B_1B_2 = \{b_1b_2|b_1 \in B_1 \wedge b_2 \in B_2\}$ denotes the set of sequences that can be formed by concatenating a sequence in B_1 with a sequence in B_2. Given a sequence b, $pref(b)$ is the set of prefixes of b and so $pref(b) = \{b'|\exists b''.b'b'' = b\}$. Given a set B_1 of sequences, $pref(B_1)$ is the set of prefixes of sequences from B_1 and so $pref(B_1) = \{b|\exists b' \in B_1.b \in pref(b')\}$.

Given a relation $f : B_1 \leftrightarrow B_2$, *dom* f denotes the set of elements from B_1 that are in the domain of f: *dom* $f = \{b_1 \in B_1|\exists b_2 \in B_2.(b_1, b_2) \in f\}$. Naturally, since every function is a relation, *dom* can also be used with functions. Given a relation $f : B_1 \leftrightarrow B_2$ and $b_1 \in B_1$, $f(b_1)$ denotes the set of elements of B_2 that are associated with b_1, i.e. $f(b_1) = \{b_2 \mid (b_1, b_2) \in f\}$.

Throughout this paper Σ denotes the set of inputs that the system (specification or IUT) can receive and Γ represents the set of outputs it can produce. We will use the following notation regarding variable names: $\sigma, \sigma', \sigma_1, \ldots$ will denote elements of Σ, $s, s', s_1, \ldots$ will denote elements of Σ^*, $\gamma, \gamma', \gamma_1, \ldots$ will denote elements of Γ, and $g, g', g_1, \ldots$ will denote elements of Γ^*.

2.2. Finite Automata

A finite automaton is defined by a finite state structure and transitions between the states, each arc between two states being labelled with an element of the finite alphabet Y.

Definition 2.1. A *finite automaton (FA) A* is defined by a tuple (Y, Q, h, q_0, T) in which Y is the finite alphabet, Q is the (non-empty) finite set of states, h is the state transfer relation of type $Q \times Y \leftrightarrow Q$, $q_0 \in Q$ is the initial state, and $T \subseteq Q$ is the set of final states.

The state transfer relation h should be interpreted in the following way: if A is in state q and receives input $y \in Y$ then it moves to one of the states in $h(q, y)$. The relation h can be extended to the relation h^* of type $Q \times Y^* \leftrightarrow Q$ defined by the following rules in which $y \in Y$ and $y' \in Y^*$: $h^*(q, \epsilon) = \{q\}$ and $h^*(q, yy') = \{q' \in Q|\exists q'' \in h(q, y).q' \in h^*(q'', y')\}$.

FA A defines a regular language: the set of sequences in Y^* that can take A from q_0 to a final state.

Definition 2.2. The FA $A = (Y, Q, h, q_0, T)$ defines the regular language $L_A = \{y \in Y^*|h^*(q_0, y) \cap T \neq \emptyset\}$. Similarly, state q of A defines the regular language $L_A(q) = \{y \in Y^*|h^*(q, y) \cap T \neq \emptyset\}$.

Two FA are *equivalent* if they define the same language. FA $A = (Y, Q, h, q_0, T)$ is *deterministic* if h is a (possibly partial) function. It is known that every FA is equivalent to a deterministic FA (DFA) and thus it is sufficient to only consider DFA (see, for example, [Eil74]). DFA A is *minimal* if no DFA with fewer states than A is equivalent to A. Since any DFA can be converted into an equivalent minimal DFA, we assume that any DFA considered in this paper is minimal.

2.3. Stream X-machines

A Stream X-machine (SXM) is essentially a FA in which there is an associated memory and each element of the alphabet represents a relation between a pair containing an input and initial memory value and a pair containing a final memory value and an output (see, for example, [Eil74, HI98]).

Definition 2.3. A stream X-machine Z is defined by a tuple $(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ in which:

- Σ is the finite input alphabet.
- Γ is the finite output alphabet.
- Q is the finite set of states.
- M is the memory.
- Φ is a set of processing relations, each having type $M \times \Sigma \leftrightarrow \Gamma \times M$.
- F is the next state function of type $Q \times \Phi \rightarrow Q$.
- $q_0 \in Q$ is the initial state.
- $m_0 \in M$ is the initial memory value.

Typically, each element of Φ specifies components that may be used in the software system specified by Z. The memory normally represents the variables used by the computer program; typically M is formed from tuples, where each element of the tuple corresponds to either a global variable or a parameter that may be passed between the elements of Φ.

Naturally, F can be extended to form a function F^* of type $Q \times \Phi^* \rightarrow Q$. If we abstract out the memory then we get a FA.

Definition 2.4. Given SXM $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, the *associated automaton* A_Z is the FA (Φ, Q, F, q_0, Q).

There are several ways in which we could generalize the definition of a SXM. First, we could use a next state relation F of type $Q \times \Phi \leftrightarrow Q$ rather than a function. It is straightforward to see that since the associated automaton A_Z must be equivalent to a DFA, we can always rewrite Z so that F is a function. Naturally, the resultant SXM need not be deterministic since the relations may not be functions. In addition, there may exist transitions, from a state s, with relations f_1 and f_2 such that there are values that satisfy the preconditions of both f_1 and f_2. We could also choose to have a set of initial states rather than a single initial state. However, if we can produce a test suite for a SXM with one initial state then we can produce a test suite for a SXM with a set I of initial states: we simply produce a test suite for each state in I and combine these test suites. Note that if more than one sequence is used then these are separated by resets and it is important that the resets always take us to the same state. In addition, if the state the reset takes us to might not be the state we start testing in then we need to start with a reset.

Each sequence in Φ^* defines a relation of type $M \times \Sigma^* \leftrightarrow \Gamma^* \times M$.

Definition 2.5. Given a sequence $p \in \Phi^*$, p defines the relation $\|p\|$, of type $M \times \Sigma^* \leftrightarrow \Gamma^* \times M$, defined by the following in which $\phi \in \Phi$, $p' \in \Phi^*$, $\sigma \in \Sigma$, $s \in \Sigma^*$, $\gamma \in \Gamma$, and $g \in \Gamma^*$.

$$\|\epsilon\| = \{((m, \epsilon), (\epsilon, m)) | m \in M\}$$

$$\begin{aligned}\|p'\phi\| = \{((m, s\sigma), (g\gamma, m')) | \exists m'' \in M.((m, s), (g, m'')) \in \|p'\| \\ \wedge((m'', \sigma), (\gamma, m')) \in \phi\}\end{aligned}$$

A machine computation takes the form of a traversal of a sequence of arcs in the state space from the initial state and the application, in turn, of the arc labels (which represent processing relations) to the initial memory value. The correspondence between the input sequence applied to the machine and the output produced gives rise to the *relation computed* by the machine.

Definition 2.6. Given a SXM Z, the *relation computed* by Z, $f_Z : \Sigma^* \longleftrightarrow \Gamma^*$, is defined by: $(s, g) \in f_Z$ if there exist $p \in L_{A_Z}$ and $m \in M$ such that $((m_0, s), (g, m)) \in \|p\|$. We say that Z *computes* f_Z.

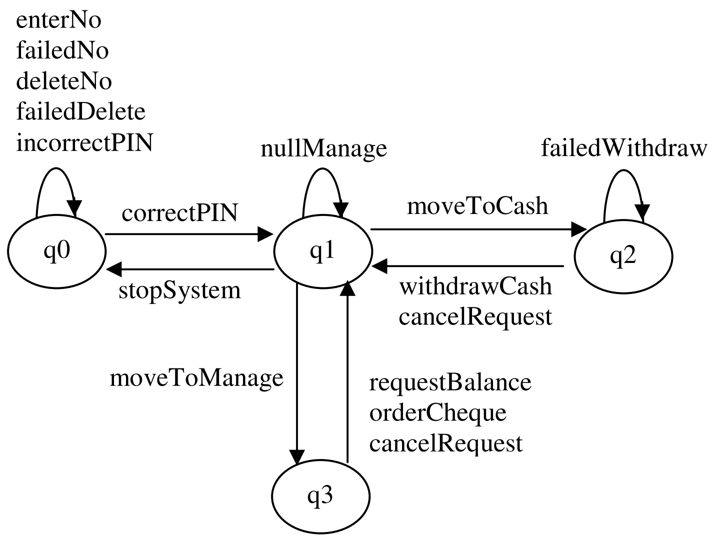

Fig. 1. The state-transition diagram of Z

The SXM Z is said to be *completely-defined* if f_Z is defined on every input sequence: $dom\ f_Z = \Sigma^*$. We only consider completely-defined SXMs in this paper. It may be possible to extend the approach to the case where Z is not completely-defined through completing Z by, for example, adding an error state.

If the SXM Z contains no infeasible paths, and so for all $p \in L_{A_Z}$ there exists some $s \in \Sigma^*$ such that $(m_0, s) \in dom\ \|p\|$, then Z is said to be *controllable*. Previous work on testing from a non-deterministic SXM has assumed that the specification is controllable, an assumption that is not made in this paper.

Traditionally the term deterministic SXM has usually been used for SXMs in which Φ is a set of functions (rather than relations) and there are no overlapping transitions emerging from the same state (if F is defined on both (q, ϕ) and (q, ϕ') with $\phi \neq \phi'$ then $dom\ \phi \cap dom\ \phi' = \emptyset$). This is a sufficient condition for a SXM to define a function from input sequences to output sequences, as opposed to a relation, but is not a necessary condition. In order to avoid confusion we use the term f-deterministic for a SXM that defines a function.

Definition 2.7. Z is said to be *f-deterministic* if f_Z is a (possibly partial) function.

Example 2.1. We now give a SXM model of a simplified ATM shown in Figure 1. In order to access the system the user must input a sequence of numbers that constitutes their PIN; to simplify the explanation the PIN will have two digits. Once one or more numbers have been provided the user can delete the most recent number. Once two numbers have been input they can press the *enter* button — at this point the actual PIN is retrieved from another system S' within the bank and compared with the PIN provided by the user. If the PIN is correct, the system moves to a state from which the user can access facilities; if not, the system returns to its initial state.

The ATM can provide two services. One service allows the customer to withdraw money from their account (if they have sufficient money in their account). A second feature is not implemented in all machines and allows the user to either obtain their current balance or order a new cheque book.

The input alphabet is the set of tuples $(button, input_{S'})$, where button is either *enter*, *delete*, or a number button $(0, \ldots, 9)$ and $input_{S'}$ is a number received from S'. When the input from S' is not needed, it is simply ignored. Similarly, each output is a tuple of the form $(message\_to\_user, screen, message\_to\_S'$, $message\_to\_ATM)$. There are four states, q_0, q_1, q_2, q_3, with q_0 being the initial state. The memory is the set of tuples of the form $(counter, pin1, pin2)$ where the elements of the tuples are numbers. The initial memory is $(0, 0, 0)$. We now describe the operations from each state.

Operations from q_0:

- *enterNo* allows the user to enter a number as part of their PIN (*counter* counts how many numbers have been provided). The guard is that the input from the user is a number button (*num*) and *counter* < 2. The output is a screen with an additional 'star'. The change in memory is defined by: $counter' = counter + 1$; if $counter = 0$ then $pin1' = num$ else $pin2' = num$. There is no change in state.

- $failedNo$ is the operation where the user attempts to enter a number but has already provided the entire PIN number. The guard is thus that the button pressed is a number (num) and $counter = 2$. The output is a message saying that PIN cannot have more than two numbers. There is no change in memory or state.
- $deleteNo$ allows the user to delete the most recent number provided. The guard is thus that the button pressed is the delete button and $counter > 0$. The output removes one 'star' from the screen and send a message that confirms that the number has been deleted. The change in memory is defined by: $counter' = counter - 1$; if $counter = 1$ then $pin1' = 0$ else $pin2' = 0$. There is no change in state.
- $failedDelete$ covers the cases where the delete button is pressed but the guard for $deleteNo$ is not satisfied. The guard is the pressing of the delete button and $counter = 0$. The output is a message saying that there is no number to delete. There is no change in memory or state.
- $correctPIN$ is the operation executed when the correct PIN has been provided and the *enter* button is pressed. The guard is the *enter* button being pressed and the PIN stored being the same as the PIN (num) sent by S': $counter = 2 \wedge num = 10 * pin2 + pin1$. The output is a new screen (menu) and a welcome message. There is no change in memory and the state becomes q_1.
- $incorrectPIN$ is the operation executed when the correct PIN has not been provided and the *enter* button is pressed. The guard is the *enter* button being pressed and the PIN stored (num) not being that sent by S': $counter < 2 \vee num \neq 10 * pin2 + pin1$. The output is a message saying that the PIN was wrong plus the removal of the 'stars' from the screen. The memory becomes $(0, 0, 0)$ and there is no change in state.

Operations from q_1:

- $moveToCash$ has the guard that the button pressed is *enter*. The output is the screen for withdrawing cash. There is no change in memory and the state becomes q_2.
- $moveToManage$ has the guard that a number button is pressed. The output is a screen for managing the account. There is no change in memory and the state becomes q_3.
- $nullManage$ has the guard that the button pressed is a number button. The output is a message saying that the ATM does not provide the 'manage account facility'. There is no change in memory or state.
- $stopSystem$ has the guard that the button pressed is *delete*. The output is a message saying that the user is being logged out and then the PIN entry screen. The memory becomes $(0, 0, 0)$ and the state becomes q_0.

Operations from q_2:

- $withdrawCash$ is triggered by input (num, bal) where the balance bal is at least the amount requested $(10 * num)$. The guard is thus: a number button (num) being pressed and $10 * num \leq bal$. The output is the menu screen associated with q_1, a message being sent to the ATM hardware to provide notes, and a message to S' confirming the amount withdrawn. There is no change in memory and the state becomes q_1.
- $failedWithdraw$ is triggered by input (num, bal) and corresponds to a failed attempt to withdraw cash. The guard is thus that a number button (num) is pressed and that $10 * num > bal$. There is no change in either memory or state.
- $cancelRequest$ allows the user to leave the state without withdrawing cash. The guard is either the *enter* or *delete* buttons being pressed. The output is the menu screen associated with q_1 and a message confirming that the option has been cancelled. There is no change in memory and the state becomes q_1.

Operations from q_3

- $cancelRequest$: as for q_2.
- $requestBalance$ is triggered by input (num, bal): the guard is that a number button (num) is pressed and num is even. The output is the menu screen for q_1 and a message giving the current balance bal. There is no change in memory and the state becomes q_1.

- *orderCheque* is triggered by input (num, bal): the guard is that a number button (num) is pressed and num is odd. The output is the menu screen for q_1, a message saying that a cheque book has been ordered, and a message to S' saying that this request has been made. There is no change in memory and the state becomes q_1.

Note that while all of the operations in the example are functions, the *withdrawCash* operation will be non-deterministic if we include information regarding different types of notes (the output is the number of each type of note that is to be provided).

2.4. Correctness

In order to test from a SXM Z we need to say what we mean by the IUT being correct. If Z is deterministic then it is normal to say that the IUT is correct if it is equivalent to Z. However, where Z is non-deterministic there is an alternative notion of correctness, called *conformance*. Here the specification defines a set of allowed output sequences for each input sequence and the IUT can choose from these. Thus the IUT conforms to Z if and only if for every input sequence s the response of the IUT to s is in $f_Z(s)$.

Definition 2.8. Z' conforms to Z if $dom\ f_{Z'} = dom\ f_Z$ and $f_{Z'} \subseteq f_Z$.

Note that the above definition is given for the general case, in which Z and Z' may not be completely-defined. As we assume that Z and Z' are completely-defined in this paper we have that $dom\ f_{Z'} = dom\ f_Z = \Sigma^*$.

In a similar manner we can say what it means for one processing relation ϕ' to conform to another processing relation ϕ.

Definition 2.9. Processing relation ϕ' of type $\Sigma \times M \leftrightarrow M \times \Gamma$ conforms to processing relation ϕ of type $\Sigma \times M \leftrightarrow M \times \Gamma$ if $dom\ \phi' = dom\ \phi$ and $\phi' \subseteq \phi$. This is denoted $\phi' \leq \phi$. Given sets Φ' and Φ of processing relations we write $\Phi' \leq \Phi$ if for all $\phi' \in \Phi$ we have some $\phi \in \Phi$ such that $\phi' \leq \phi$.

3. Design for test conditions and test hypotheses

There are conditions associated with the use of SXMs for specification and design, the intention of these conditions being to make testing tractable. These conditions can be split into design for test conditions that place restrictions on the specification Z and test hypotheses that place restrictions on the IUT Z'.

3.1. Design for test conditions on the specification

In testing we observe input/output sequences. We would like to be able to identify the corresponding sequences of relations that were executed and in order to achieve this we insist that two relations cannot produce the same output for a given memory and input.

Definition 3.1. Φ is *output distinguishable* if for all $\phi_1, \phi_2 \in \Phi$ such that $\phi_1 \neq \phi_2$, all $\sigma \in \Sigma$, all $\gamma \in \Gamma$, and all $m, m' \in M$ such that $((m, \sigma), (\gamma, m')) \in \phi_1$, there does not exist $m'' \in M$ such that $((m, \sigma), (\gamma, m'')) \in \phi_2$.

Let us suppose that we applied the input sequence s, observed output sequence g and wish to now apply an input to determine whether the relation ϕ_1 can be triggered from the state of Z' reached. In order to choose an appropriate input σ we need to know the current memory m since we require that $(\sigma, m) \in dom\ \phi_1$. If Φ is output distinguishable then we can determine the sequence of relations that corresponds to s/g. Thus, in order to determine the memory after s/g it is sufficient that for all $\phi \in \Phi$ we have that the memory after ϕ has been applied is fully determined by the memory before ϕ was applied, the input used, and the output observed.

Definition 3.2. Φ is *observable* if for all $\phi \in \Phi, m \in M, \sigma \in \Sigma$

$$(\gamma_1, m_1), (\gamma_2, m_2) \in \phi(m, \sigma) \Rightarrow ((\gamma_1 = \gamma_2) \Rightarrow (m_1 = m_2)).$$

If all sequences in L_{A_Z} are feasible then Z is said to be controllable and naturally this significantly simplifies test sequence generation. Previous work on testing from a non-deterministic SXM has assumed that Φ is controllable. However, this is a strong restriction and many specifications are not controllable. Instead we make a weaker assumption, recently introduced for testing from a deterministic SXM [Ipa06].

Informally, Φ is *input-uniform* if for all $p \in \Phi^*$, all memory values that can be produced by the application of p when applied in a given memory m are processed in a uniform way by all $\phi \in \Phi$: if $\phi \in \Phi$ then either ϕ can process all such memory values or none. The memory values that can be produced as the result of applying a sequence of processing relations from m_0 will be said to be *image-similar*. The memory values that are processed uniformly by all $\phi \in \Phi$ will be said to be *domain-similar*.

Definition 3.3. Two memory values $m_1, m_2 \in M$ are *domain-similar* if for all $\phi \in \Phi$, there exists $\sigma_1 \in \Sigma$ such that $(m_1, \sigma_1) \in dom\ \phi$ if and only if there exists $\sigma_2 \in \Sigma$ such that $(m_2, \sigma_2) \in dom\ \phi$.

Naturally domain-similarity is an equivalence relation.

Definition 3.4. $\mathcal{I}M_j$, $j \geq 0$, are relations on M defined as follows:

- $(m, m) \in \mathcal{I}M_0$, $m \in M$.
- Given $j > 0$ and $m_1, m_2 \in M$, $(m_1, m_2) \in \mathcal{I}M_j$ if:
  - $(m_1, m_2) \in \mathcal{I}M_{j-1}$; or
  - there exist $m_1', m_2' \in M$ such that $((m_1', m_2') \in \mathcal{I}M_{j-1}$ and there exist $\phi \in \Phi$, $\sigma_1, \sigma_2 \in \Sigma$, $\gamma_1, \gamma_2 \in \Gamma$, such that $((\gamma_1, m_1) \in \phi(m_1', \sigma_1)$ and $(\gamma_2, m_2) \in \phi(m_2', \sigma_2)))$.

Memory values $m_1, m_2 \in M$ are *image-similar* if $(m_1, m_2) \in \mathcal{I}M_j$ for some $j \geq 0$.

A direct consequence of the above definition is that if there exists $j \geq 0$ such that $\mathcal{I}M_j = \mathcal{I}M_{j+1}$ then for all $i \geq 1$ we have that $\mathcal{I}M_j = \mathcal{I}M_{j+i}$ and so the image-similarity relation coincides with $\mathcal{I}M_j$. In essence two memory values $m_1, m_2 \in M$ are image-similar if there is some $p \in \Phi^*$, a memory value $m \in M$ and input sequences s_1 and s_2 such that we can obtain memory m_i after executing p with s_i given initial memory m.

Note that image-similarity, as defined above, is not a transitive relation. On the other hand, its transitive closure could have been used instead, without affecting the definition of input-uniformity (Definition 3.5).

Definition 3.5. Φ is called *input-uniform* if for all $m_1, m_2 \in M$, if m_1 and m_2 are image-similar then m_1 and m_2 are domain-similar.

When Φ is input-uniform and one is trying to drive a sequence of processing relations in testing then it is possible to apply an iterative process in which input symbols are selected one at a time: there is no need to consider the relations that are to be applied after the one currently being considered.

We can now express the overall design for test conditions on the specification.

Definition 3.6. Given a SXM $Z = (\Sigma, \Gamma, Q, M, \Phi, F', q_0, m_0)$ the *design for test conditions* are that Φ is:

- output-distinguishable;
- observable; and
- input-uniform.

Now consider the example given in Figure 1. The SXM machine is non-deterministic due to the account management features being optional. It clearly is not controllable, for example the sequence *deleteNo* is not feasible since in order to apply this we first need to provide at least one number. However, it is straightforward to show that it is input-uniform. The different messages sent by the operations ensure that they are output-distinguishable. All of the operations are functions and so are automatically observable. As noted earlier, *withdrawCash* is non-deterministic if we add information about the notes produced when this operation is used. However, the operation would still be observable since the notes left in the machine after cash is withdrawn is fully determined by the notes in the machine before this operation and the notes given to the customer.

Recall that we assume that Z is completely-defined. We also assume that certain test hypotheses hold.

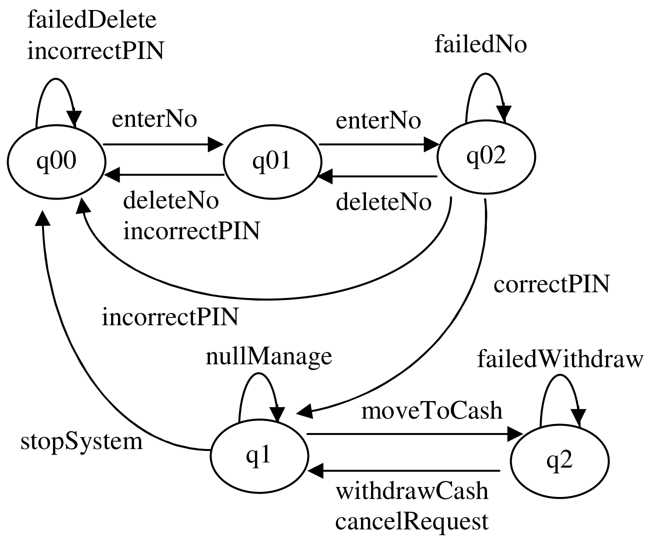

Fig. 2. The state-transition diagram of Z_1'

3.2. Test hypotheses on the implementation

In order to reason about test effectiveness it is normal to assume that the IUT is functionally equivalent to some unknown element of a given fault domain that contains a set of models (see, for example, [IT97, RMN06]). When testing from an SXM Z this fault domain contains SXMs with the same memory[1], initial memory, input alphabet and output alphabet as Z. Thus, since the IUT is deterministic, we assume that the IUT behaves like an unknown f-deterministic SXM $Z' = (\Sigma, \Gamma, Q', M, \Phi', F', q_0', m_0)$. In common with many approaches for testing from other state-based models, such as finite state machines, we assume that there is a known upper bound n' on the number of states of Z'.

Definition 3.7. Given a SXM $Z = (\Sigma, \Gamma, Q, M, \Phi, F', q_0, m_0)$ with n states the *test hypotheses* are that the IUT behaves like an unknown controllable, completely-defined, f-deterministic SXM $Z' = (\Sigma, \Gamma, Q', M, \Phi', F', q_0', m_0)$ that has at most n' states (some given n') such that $\Phi' \leq \Phi$.

The *fault domain* thus contains all such f-deterministic SXMs. Observe that while we do not assume that the specification Z is controllable we do assume that the implementation Z' is controllable. However, since in practice the memory is finite, there is always a controllable stream X-machine that models the IUT. The assumption that $\Phi' \leq \Phi$ corresponds to Φ containing the specifications of the components in Φ' and these being trusted components since it requires that each component $\phi' \in \Phi'$ in the IUT conforms to a component $\phi \in \Phi$ in the specification.

In our example, a controllable model of Z can be obtained by creating distinct states, $q00$, $q01$, $q02$ for each value taken by the counter when the PIN is entered. Then, by removing the non-determinism caused by the overlapping domains of $moveToManage$ and $nullManage$, two controllable SXMs, Z_1' and Z_2', that conform to Z are obtained. These are represented in Figure 2 and Figure 3, respectively. Thus the test generation problem can be formulated for $n' \geq 5$.

From the design for test conditions and the test hypotheses (the condition $\Phi' \leq \Phi$) it follows that

- Φ' is also observable and input-uniform and
- every processing relation in the IUT corresponds to an unique processing relation in the specification.

Lemma 3.1. Let us suppose the Φ is observable, input-uniform, and output-distinguishable. If $\Phi' \leq \Phi$ then Φ' is observable and input-uniform.

[1] We assume that the memory of Z' is known since the role of the memory is to allow values to be passed between relations and thus is defined by the interfaces of the (trusted) components used in development.

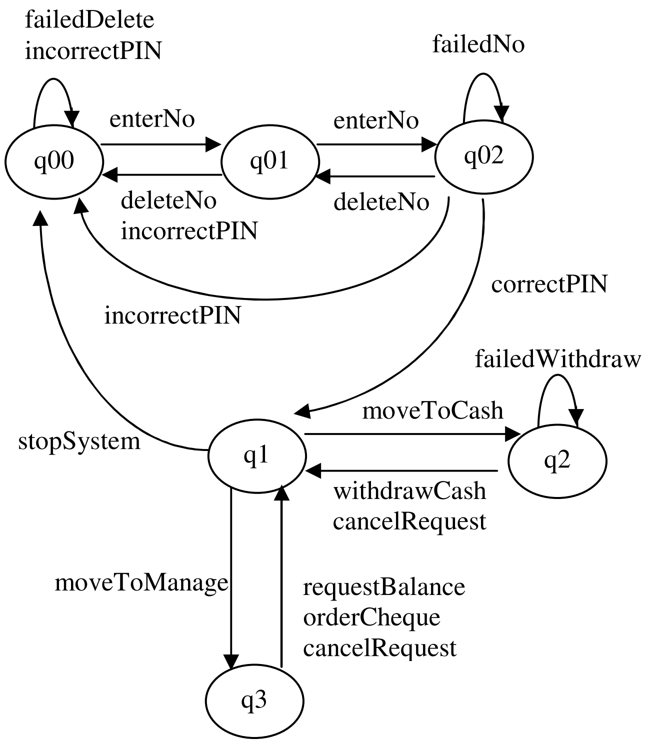

Fig. 3. The state-transition diagram of Z'_2

Proof. We first prove that Φ' is observable. Let us suppose that $\phi' \in \Phi'$, $(\gamma_1, m_1), (\gamma_2, m_2) \in \phi'(m, \sigma)$. Then we require to prove that $(\gamma_1 = \gamma_2) \Rightarrow (m_1 = m_2)$. Since $\Phi' \leq \Phi$ there exists some $\phi \in \Phi$ such that $\phi' \leq \phi$. Thus, $(\gamma_1, m_1), (\gamma_2, m_2) \in \phi(m, \sigma)$. The result now follows from Φ being observable.

Now let us assume that memory values m_1 and m_2 are image-similar for Φ': we require to prove that they are domain-similar for Φ'. Since $\Phi' \leq \Phi$ it is clear that m_1 and m_2 must be image-similar for Φ. Since Φ is input-uniform, m_1 and m_2 must be domain-similar for Φ. Further, since $\Phi' \leq \Phi$, if m_1 and m_2 are domain-similar for Φ then they are domain-similar for Φ'. The result thus follows. □

Lemma 3.2. Let us suppose the Φ is observable and output-distinguishable. If $\Phi' \leq \Phi$ then for every $\phi' \in \Phi'$ there is exactly one $\phi \in \Phi$ such that $\phi' \leq \phi$.

Proof. Proof by contradiction: assume that there exists some $\phi' \in \Phi'$ and $\phi_1, \phi_2 \in \Phi$ such that $\phi' \leq \phi_1$, $\phi' \leq \phi_2$, and $\phi_1 \neq \phi_2$. By definition, $dom\ \phi_1 = dom\ \phi_2 = dom\ \phi'$. Choose some $(m, \sigma) \in dom\ \phi'$ and some $(\gamma_1, m_1) \in \phi'(m, \sigma)$. Since $\phi' \leq \phi_1$ we must have that $(\gamma_1, m_1) \in \phi_1(m, \sigma)$. Similarly, $(\gamma_1, m_1) \in \phi_2(m, \sigma)$. But this contradicts Φ being output distinguishable. □

$Abs(Z') = (\Sigma, \Gamma, Q', M, \Phi, F'_{abs}, q'_0, m_0)$ will denote the abstraction of Z', formed by replacing each $\phi' \in \Phi'$ by the unique relation $\phi \in \Phi$ such that $\phi' \leq \phi$. This relation ϕ will be denoted $abs(\phi')$.

Throughout this paper we assume that the design for test conditions and test hypotheses hold: the problem now is to show how an automated process can produce a test suite that determines correctness relative to these conditions.

4. Reaching and distinguishing states in a SXM

Test generation algorithms, for testing from a SXM or an FSM, typically use sequences that reach states and distinguish the specification: these are used to explore the state structure of the IUT. However, as a SXM Z may have infeasible paths, there might be states that are reachable in the diagram that cannot be reached by any input sequence applied to the machine. Similarly, there may be pairs of distinguishable states in the associated automaton for which the sequences of processing relations that distinguish between them can never be applied (see [Ipa06] for an example). In this section we therefore introduce terminology and notation regarding feasible paths in a SXM.

4.1. Realisable sequences

In order to determine which states can actually be reached or distinguished, we have to establish which sequences of processing relations in the associated automaton can be driven by input sequences. Such sequences of processing relations are said to be *realisable*.

Definition 4.1. Given a memory value $m \in M$, the set $R_\Phi(m)$ consists of all sequences of processing relations $p = \phi_1 \ldots \phi_n \in \Phi^*$, $n \geq 0$, for which there exists $s = \sigma_1 \ldots \sigma_n \in \Sigma^*$ such that $(m, s) \in dom\ \|p\|$.

Definition 4.2. Given a state $q \in Q$ and a memory value $m \in M$, a sequence of processing relations $p \in \Phi^*$ is said to be *realisable* in q and m if $p \in L_{A_Z}(q)$ and $p \in R_\Phi(m)$. If $q = q_0$ and $m = m_0$, p is simply said to be realisable. The set of all sequences of processing relations realisable (in q and m) is denoted by LR_Z (or $LR_Z(q, m)$) and so $LR_Z(q, m) = L_{A_Z}(q) \cap R_\Phi(m)$ and $LR_Z = L_{A_Z} \cap R_\Phi(m_0)$.

If all sequences of processing relations accepted by the associated automaton are realisable then the SXM is said to be controllable.

Definition 4.3. Z is said to be *controllable* if $LR_Z = L_{A_Z}$.

Due to non-determinism, an input sequence s may not always drive a sequence p from a memory value m even if (m, s) is contained in the domain of p. This may happen where the domain of p intersects the domain of other sequences from $LR_Z(q, m)$. However, given a state q and a memory value m of Z, it may be possible to identify sequences that are guaranteed to be always driven by input sequences. Such sequences of processing relations are called *deterministically-realisable* (*d-realisable*). As d-realisable sequences process input sequences that are not processed by any other path in the specification Z, they must be in any implementation that conforms to Z. They can therefore be used in testing to reach states of the IUT.

Definition 4.4. Given a state $q \in Q$ and a memory value $m \in M$, a sequence of processing relations $p \in LR_Z(q, m)$ is said to be *d-realisable* in q and m if for all $s \in \Sigma^*$ such that $(m, s) \in dom\ \|p\|$, the following holds: $(m, s) \notin \bigcup_{x \in LR_Z(q,m) \setminus \{p\}} dom\ \|x\|$. If $q = q_0$ and $m = m_0$, p is simply said to be d-realisable. The set of all sequences of processing relations that are d-realisable (in q and m) is denoted by LDR_Z (or $LDR_Z(q, m)$).

Interestingly, the above condition may be weakened: it is sufficient that for a state q and a memory value m there is some such input sequence s. Such a definition might state that a sequence p is contained in $LDR_Z(q, m)$ if and only if *there exists* $s \in \Sigma^*$ such that $(m, s) \in dom\ \|p\|$ and the following holds: $(m, s) \notin \bigcup_{x \in LR_Z(q,m) \setminus \{p\}} dom\ \|x\|$. However, if the weaker condition is used then the test process (Definition 6.1), that associates an input/output sequence to every sequence of processing function, will have to be defined in a more complex manner. The above, stronger, condition, will be used throughout this paper in order to aid readability, but, in practice, the weaker condition may lead to a more efficient test generation procedure.

4.2. dr-reachable states

Sequences in LDR$_Z$ make it possible to reach some states of a SXM using appropriate input sequences and we know that they should reach corresponding states of the IUT. Such states will be referred to as *dr-reachable*. Since $\epsilon \in \text{LDR}_Z$, the initial state is always dr-reachable.

Definition 4.5. A state $q \in Q$ is said to be *dr-reachable* if there exists $p \in LDR_Z$ such that $F^*(q_0, p) = q$.

An dr-state cover is a minimal set of realisable sequences S_{dr}, $\epsilon \in S_{dr}$, that reaches every dr-reachable state in Z.

Definition 4.6. A set $S_{dr} \subseteq LDR_Z$ is a *dr-state cover* of Z if:

- $\epsilon \in S$.
- For every dr-reachable state q of Z there exists $p \in S_{dr}$ such that $F^*(q_0, p) = q$.
- For every two distinct sequences $p_1, p_2 \in S_{dr}$, $F^*(q_0, p_1) \neq F^*(q_0, p_2)$.

Ideally, a dr-state cover will be used in test generation. However, the test generation algorithm will use a set of sequences that dr-reach states of the SXM specification but will not require the use of a dr-state cover.

For Z as in Example 2.1, the sequences ϵ, $enterNo\ enterNo\ correctPIN$, and $enterNo\ enterNo\ correctPIN\ moveToCash$ are all in LDR$_Z$, so q_0, q_1 and q_2 are dr-reachable. On the other hand, q_3 is not dr-reachable since it can only be reached (from q_2) by $moveToManage$ and $dom\ moveToManage = dom\ nullManage$. Thus $S_{dr} = \{\epsilon, enterNo\ enterNo\ correctPIN, enterNo\ enterNo\ correctPIN\ allowbreakmoveToCash\}$ is a dr-state cover of Z.

4.3. Attainable memory values

The memory values computed along sequences in LR$_Z$ that reach a state q will be said to be *attainable* in q.

Definition 4.7. Given a state $q \in Q$, a memory value $m \in M$ is said to be *attainable* in q if there exist $p \in LR_Z$, $s \in \Sigma^*$, $g \in \Gamma^*$ such that $F^*(q_0, p) = q$ and $((m_0, s), (g, m)) \in \|p\|$. The set of all memory values attainable in q is denoted by $MAtt(q)$.

4.4. dr-distinguishable states

In test generation it is normal to use sequences that distinguish the states of the specification: these should also distinguish the corresponding states of the IUT. We will say that two states q_1 and q_2 of a SXM are dr-distinguishable if it is possible to distinguish between them by applying a finite set of d-realisable sequences of processing relations in any attainable memory value of q_1 and q_2, respectively.

Definition 4.8. Given $q_1, q_2 \in Q$, a set $Y \subseteq \Phi^*$ is said to *dr-distinguish* between q_1 and q_2 if for every $m_1 \in MAtt(q_1)$ and every $m_2 \in MAtt(q_2)$, there exists $p \in Y$ such that $p \in LDR_Z(q_1, m_1) \setminus LR_Z(q_2, m_2)$ or $p \in LDR_Z(q_2, m_2) \setminus LR_Z(q_1, m_1)$. Two states q_1 and q_2 are said to be *dr-distinguishable* if there exists a *finite* set of sequences Y that dr-distinguishes between them.

In other words, states q_1 and q_2 are dr-distinguishable if there exists a finite set of sequences Y such that for every memory values m_1 and m_2, attainable in q_1 and q_2, respectively, Y contains a path p that is d-realisable in q_1 and m_1 and not realisable in q_2 and m_2 or vice versa.

When Z is f-deterministic, LDR_Z coincides with LR_Z, so the above condition becomes $LDR_Z(q_1, m_1) \cap Y \neq LDR_Z(q_2, m_2) \cap Y$. This coincides with the definition of r-distinguishable states given in [Ipa06] for deterministic SXMs. As shown in [Ipa06], not every pair of states of a SXM can necessarily be dr-distinguished (r-distinguished) by a set of sequences even if the associated FA is minimal and, furthermore, even if such a set exists, it may not be finite.

A dr-characterization set is a set of sequences of processing relations that dr-distinguishes between every pair of dr-distinguishable states.

Definition 4.9. A set $W_{dr} \subseteq \Phi^*$ is called a *dr-characterization set* of Z if W_{dr} dr-distinguishes between every two dr-distinguishable states of Z.

Ideally, we use a dr-characterization set in test generation. However, the test generation algorithm will use a set W of sequences that dr-distinguish states of the specification SXM but will not require the use of a dr-characterization set. Additionally, if we have no sequences that dr-distinguish states of the specification, W will just contain the empty sequence.

For Z as in Example 2.1, it can be observed that, for any memory value $m \in M$, $moveToCash \in LDR(q_1, m)$, $withdrawCash \in LDR(q_2, m)$ (we assume that for every number entered by the user, S' can provide an appropriate balance that meets the triggering condition) and $requestBalance \in LDR(q_3, m)$. Thus, all states of Z are pairwise dr-distinguishable and $W_{dr} = \{moveToCash, withdrawCash, requestBalance\}$ is a dr-characterization set of Z.

4.5. Checking realisable sequences

As Z and $Abs(Z')$ are assumed to have the same type and this is output-distinguishable and observable, the testing process involves checking that the sequences of processing relations allowed by the implementation are contained in the set specified. Only the realisable sequences have to be considered, as the others have no functional role. This idea is captured by the following lemma.

Lemma 4.1. *Z' conforms to Z if and only if $LR_{Abs(Z')} \subseteq LR_Z$.*

Proof. "$\Leftarrow$": From Definition 2.6 it follows that $Abs(Z')$ conforms to Z. Since $\Phi' \leq \Phi$, Z' conforms to $Abs(Z')$ and so Z' conforms to Z.

"$\Rightarrow$": Let $p' = \phi'_1 \dots \phi'_k \in LR_{Abs(Z')}$ and let $\phi_i = abs(\phi'_i)$, $1 \leq i \leq k$. Then there exist $\sigma_1, \dots, \sigma_k \in \Sigma$, $\gamma_1, \dots, \gamma_k \in \Gamma$, $m_1, \dots, m_k \in M$ such that $((m_{i-1}, \sigma_i), (\gamma_i, m_i)) \in \phi'_i$, $1 \leq i \leq k$. Then $((m_{i-1}, \sigma_i), (\gamma_i, m_i)) \in \phi_i$, $1 \leq i \leq k$. Since Z' conforms to Z there exist $\phi''_1, \dots, \phi''_k \in \Phi$, $m''_1, \dots, m''_k \in M$ such that $\phi''_1 \dots \phi''_k \in LR_Z$ and $(\gamma_i, m''_i) \in \phi''_i(m''_{i-1}, \sigma_i)$, $1 \leq i \leq k$, where $m''_0 = m_0$. Since Φ is output-distinguishable and observable, by induction on i, $1 \leq i \leq k$, it follows that $\phi''_i = \phi_i$ and $m''_i = m_i$. Thus $p \in LR_Z$. Since p is arbitrarily chosen, $LR_{Abs(Z')} \subseteq LR_Z$. $\square$

Consequently, since Z' is controllable, it is sufficient to check that every sequence of processing relations in the associated FA of $Abs(Z')$ is also accepted by the associated FA of Z.

Lemma 4.2. *$LR_{Abs(Z')} \subseteq LR_Z$ if and only if $L_{A_{Abs(Z')}} \subseteq L_{A_Z}$.*

Proof. "$\Leftarrow$": Assume $L_{A_{Abs(Z')}} \subseteq L_{A_Z}$. Then $L_{A_{Abs(Z')}} \cap R_\Phi(m_0) \subseteq L_{A_Z} \cap R_\Phi(m_0)$. Thus $LR_{Abs(Z')} \subseteq LR_Z$.

"$\Rightarrow$": Conversely, assume $LR_{Abs(Z')} \subseteq LR_Z$. Then $L_{A_{Abs(Z')}} \cap R_\Phi(m_0) \subseteq L_{A_Z} \cap R_\Phi(m_0)$. Since Z' is controllable, $L_{A_{Abs(Z')}} \cap R_\Phi(m_0) = L_{A_{Abs(Z')}}$, so $L_{A_{Abs(Z')}} \subseteq L_{A_Z} \cap R_\Phi(m_0)$. Thus $L_{A_{Abs(Z')}} \subseteq L_{A_Z}$. $\square$

5. The product machine

A state-counting approach will be used in order to establish whether $L_{A_{Abs(Z')}} \subseteq L_{A_Z}$. The reasoning behind this involves the product machine of Z and $Abs(Z')$ and this will now be defined. State-counting was originally used for conformance testing of a deterministic implementation against a non-deterministic finite state machine [PYB96] and has been more recently applied to test generation from stream X-machines [HH04]. We will be able to express conformance of the IUT to the specification SXM in terms of the reachability of a state $Fail$ in the product machine and test generation will be based on this.

Given two FA, A_Z and $A_{Abs(Z')}$, it is possible to build a cross-product of their states, such that states (q, q') of the cross-product FA correspond to pairs of states q, q' in the two FA. A transition $F_P((q, q'), \phi) = (q_1, q'_1)$ exists in the cross-product FA if and only if the transitions $F(q, \phi) = q_1$ and $F'_{abs}(q', \phi) = q'_1$ exist in A_Z and $A_{Abs(Z')}$, respectively. The language accepted by the automaton produced by this construction corresponds to the intersection of the languages accepted by the two FAs. If the language accepted by $A_{Abs(Z')}$ is not contained in that accepted by A_Z, then there will be a transition from some (q, q') that $Abs(Z')$ can follow but Z cannot. We add to the cross-product FA an extra state, $Fail$, and transitions $F_P((q, q'), \phi) = Fail$ to correspond to transitions that can be taken by $Abs(Z')$ but not by Z. Testing for inclusion of $L_{A_{Abs(Z')}}$ in L_{A_Z} then corresponds to testing in order to determine whether the state $Fail$ of the cross-product FA is reachable. If the two SXMs, Z and $Abs(Z')$, are considered instead of their FA, this construction defines the *product machine*.

Definition 5.1. The *product machine* formed from $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $Abs(Z') = (\Sigma, \Gamma, Q', M, \Phi, F'_{abs}, q'_0, m_0)$ is the SXM $P(Z, Abs(Z')) = (\Sigma, \Gamma, Q_P, M, \Phi, F_P, (q_0, q'_0), m_0)$ in which $Q_P = (Q \times Q') \cup \{Fail\}$, $Fail \notin Q \times Q'$, and F_P is defined by the following rules:

- For $(q, q') \in Q_P$ and $\phi \in \Phi$:
  - If $(q, \phi) \in dom\ F$ and $(q', \phi) \in dom\ F'_{abs}$ then $F_P((q, q'), \phi) = (F(q, \phi), F'_{abs}(q', \phi))$.
  - If $(q, \phi) \notin dom\ F$ and $(q', \phi) \in dom\ F'_{abs}$ then $F_P((q, q'), \phi) = Fail$.
  - Else $F_P((q, q'), \phi)$ is undefined.

- For $\phi \in \Phi$, $F_P(Fail, \phi)$ is undefined.

Since Z' is controllable and every sequence of processing relations accepted by $A_{P(Z,Abs(Z'))}$ is also accepted by $A_{Abs(Z')}$, $P(Z, Abs(Z'))$ is also controllable. Note also that, unlike Z and Z', $P(Z, Abs(Z'))$ need not be completely-defined. This is not a problem, however, since it will be sufficient to check whether the $Fail$ state is reachable.

The remainder of this section shows that determining whether Z' conforms to Z corresponds to determining whether the $Fail$ state of the product machine is reachable.

Lemma 5.1. *Given $p \in \Phi^*$, $q \in Q$, $q' \in Q'$, $F_P^*((q_0, q_0'), p) = (q, q')$ if and only if $F^*(q_0, p) = q$ and $F_{abs}'^*(q_0', p) = q'$.*

Proof. Follows from Definition 5.1 by induction on the length of p. □

Lemma 5.2. *Given $p \in \Phi^*$, p reaches $Fail$ in $A_{P(Z,Abs(Z'))}$ if and only if $p \in L_{A_{Abs(Z')}} \setminus L_{A_Z}$ and $p = p_1\phi$ for some $p_1 \in L_{A_Z} \cap L_{A_{Abs(Z')}}$ and $\phi \in \Phi$.*

Proof. By Definition 5.1, p reaches $Fail$ in $\mathrm{A}_{P(Z,Abs(Z'))}$ if and only if $p = p_1\phi$ for some $p_1 \in \Phi^*$, $\phi \in \Phi$ for which there exist $q \in Q$, $q' \in Q'$ such that $F_P^*((q_0, q_0'), p_1) = (q, q')$ and $F_P((q, q'), \phi) = Fail$. By Lemma 5.1, $F_P^*((q_0, q_0'), p_1) = (q, q')$ if and only if $F^*(q_0, p_1) = q$, $F_{abs}'^*(q_0', p_1) = q'$. By Definition 5.1, $F_P((q, q'), \phi) = Fail$ if and only if $(q, \phi) \notin dom\ F$ and $(q', \phi) \in dom\ F'_{abs}$. Thus, p reaches $Fail$ in $A_{P(Z,Abs(Z'))}$ if and only if $p \in L_{A_{Abs(Z')}} \setminus L_{A_Z}$ and $p = p_1\phi$ for some $p_1 \in L_{A_Z} \cap L_{A_{Abs(Z')}}$ and $\phi \in \Phi$. □

Lemma 5.3. *$Fail$ is not reachable in $A_{P(Z,Abs(Z'))}$ if and only if $L_{A_{Abs(Z')}} \subseteq L_{A_Z}$.*

Proof. $L_{A_{Abs(Z')}} \subseteq L_{A_Z}$ does not hold if and only if there exist $p \in \Phi^*$, $\phi \in \Phi$ such that $p \in L_{A_Z} \cap L_{A_{Abs(Z')}}$ and $p\phi \in L_{A_{Abs(Z')}} \setminus L_{A_Z}$. Thus, by Lemma 5.2, $Fail$ is reachable in $A_{P(Z,Abs(Z'))}$ if and only if $L_{A_{Abs(Z')}} \subseteq L_{A_Z}$ does not hold. □

Lemma 5.4. *$Fail$ is not reachable in $A_{P(Z,Abs(Z'))}$ if and only if $LR_{Abs(Z')} \subseteq LR_Z$.*

Proof. Follows from Lemmas 5.3 and 4.2. □

Lemma 5.5. *$Fail$ is not reachable in $A_{P(Z,Abs(Z'))}$ if and only if Z' conforms to Z.*

Proof. Follows from Lemmas 5.4 and 4.1. □

6. Test processes

Let us suppose that we have generated appropriate sequences of processing relations to check whether the $Fail$ state of $A_{P(Z,Abs(Z'))}$ is reachable. We will then need a mechanism, called a *test process* of Z that translates each sequence of processing relations into a pair containing an input sequence and the corresponding output sequence observed in testing. In effect, the test process is used in order to determine in testing whether a path in the SXM specification has been implemented. We require a test process, rather than a preset input sequence, because of the nondeterminism in the SXM specification. The concept of a test process was originally defined for a (controllable) quasi-nondeterministic SXM [HH00] and then the definition was extended to any (controllable) non-deterministic SXM [HH04]. We further extend this definition to the case in which Φ is input-uniform but may not be controllable.

A test process takes a sequence $p \in \Phi^*$ of relations and attempts to find test data to execute p. It does this in an adaptive manner: if a prefix p' of p has been successfully executed then it finds an input that should trigger the next relation given the current memory. If at any point the test process fails to trigger the expected relation then it terminates. Note that since Φ is output-distinguishable we know whether a relation conforming to a required relation ϕ has been triggered. In addition, since Φ is observable we know the current memory after an input/output sequence has been observed.

Definition 6.1. Let us suppose that Φ is input-uniform. A *test process* of Z is a function $t : \Phi^* \longrightarrow \Sigma^* \times \Gamma^*$ that satisfies the following conditions:

- $t(\epsilon) = \epsilon$. (1)

- Let $p \in \Phi^*$ and $\phi \in \Phi$ and let $t(p) = (s, g)$.
  - Assume that $p \in L_{A_Z}$ and there exists $m \in M$ such that $((m_0, s), (g, m)) \in \|p\|$.
    - If there exists $\sigma \in \Sigma$ such that $(m, \sigma) \in dom\ \phi$ then $t(p\phi) = t(p)(\sigma, \gamma)$ for some σ that satisfies this condition and γ such that Z' produces γ in response to σ after (s, g). (2)
    - Else, $t(p\phi) = t(p)$. (3)
  - Otherwise, $t(p\phi) = t(p)$. (4)

The first rule is the base case, stating that testing based on the empty sequence requires no input and produces no output. The remaining three rules are recursive cases, explaining how the test for sequence $p\phi$ is defined in terms of $t(p)$. The second and third rules give the case where p has been triggered in $Abs(Z')$: if an appropriate input can be found in the domain of ϕ, the sequence is extended further (rule 2); otherwise, the sequence cannot be extended and the construction of $t(p\phi)$ reduces to the construction of $t(p)$ (rule 3). The final rule states how a sequence p is pruned.

Pruning happens in the following two cases: either some prefix of p is not contained in L_{A_Z} or $t(p)$ triggers some other sequence in $Abs(Z')$. In this paper the test process is used to decide whether $L_{A_{Abs(Z')}}$, the language defined by the abstraction of the implementation machine is included in L_{A_Z}, the language defined by the specification. If a prefix of p is not contained in L_{A_Z} then p is not in L_{A_Z} and so there is no need to test further. Similarly, if $t(p)$ triggers some sequence $p' \neq p$ in $Abs(Z')$ then, since Z is f-deterministic and Φ is output-distinguishable, we can deduce that p is not contained in $L_{A_{Abs(Z')}}$. Then it is not necessary for the test process to test beyond p: it is sufficient to establish that $p \notin L_{A_{Abs(Z')}}$. Note that the IUT Z' is an implicit parameter of the test process t.

Let us suppose that the test process is applied to a sequence $p = \phi_1, \ldots, \phi_k$ from L_{A_Z}. The test process follows a sequence of steps. At the ith step, the test process produces an input σ_i that can trigger ϕ_i, given the current memory. The input σ_i is sent to the IUT and the output is observed. From this, since Φ is observable and output-distinguishable, the memory after the transition can be determined. The next input used depends upon the output received in response to previous input since Z is non-deterministic: the choice of next input depends upon the current memory. Since there may be more than one acceptable input at some point, there can be more than one possible test process.

In general, the test process associates a sequence of processing relations p with a pair (s, g), where g represents the output produced by the application of s to Z'. This is shown by Lemma 6.1. A direct consequence of this result is that, whenever Z' conforms to Z, the test process will produce only input/output pairs that are allowed by the specification.

Lemma 6.1. Let us suppose that Φ is input-uniform, output-distinguishable and observable. Let $t : \Phi^* \longrightarrow \Sigma^* \times \Gamma^*$ be a test process of Z, $p \in \Phi^*$ and let $(s, g) = t(p)$. Then there exist $p' \in LR_{Abs(Z')}$ and unique $m \in M$ such that $((m_0, s), (g, m)) \in \|p'\|$.

Proof. We prove the result by induction on the length of p. The result clearly holds for the base case ϵ.

Let us suppose that the result holds for p and let $\phi \in \Phi$. Suppose $p \in L_{A_Z}$, there exists $m \in M$ such that $((m_0, s), (g, m)) \in \|p\|$. By the inductive hypothesis, there exists $p' \in LR_{Abs(Z')}$ and unique $m \in M$ such that $((m_0, s), (g, m)) \in \|p'\|$. As Φ is output-distinguishable and observable, $p' = p$, so $p \in LR_{Abs(Z')}$. If there exists $\sigma \in \Sigma$ such that $(m, \sigma) \in dom\ \phi$ then $t(p\phi) = (s\sigma, g\gamma)$ for some σ and γ for which there exist $\phi' \in \Phi'$ and $m' \in M$ such that $p\phi' \in LR_{Abs(Z')}$ and $((m, \sigma), (\gamma, m')) \in \phi'$. As Φ is observable, m' is unique and so the result follows. In any other case, $t(p\phi) = (s, g)$ and the result follows directly from the inductive hypothesis. $\square$

On the other hand, the test process may be used to explore the relationship between $L_{A_{Abs(Z')}}$, the language defined by the abstraction of the implementation, and L_{A_Z}, the language defined by the specification, as shown by the following result.

Lemma 6.2. Let us suppose that Φ is input-uniform, output-distinguishable and observable. Let $t : \Phi^* \longrightarrow \Sigma^* \times \Gamma^*$ be a test process of Z, $p \in \Phi^*$ and let $(s, g) = t(p)$. If $p \in LR_{Abs(Z')}$ and $(s, g) \in f_Z$ then $p \in L_{A_Z}$ and there exists unique $m \in M$ such that $((m_0, s), (g, m)) \in \|p\|$.

Proof. We prove the result by induction on the length of p. The result clearly holds for the base case ϵ.

Let us suppose that the result holds for p and let $\phi \in \Phi$ such that $p\phi \in LR_{Abs(Z')}$. By the inductive hypothesis, $p \in L_{A_Z}$ and there exists unique $m \in M$ such that $((m_0, s), (g, m)) \in \|p\|$. Since Φ is input-uniform and $p\phi \in R_\Phi(m_0)$, there exists $\sigma \in \Sigma$ such that $(m, \sigma) \in dom\ \phi$. Then, by rule (2) of Definition 6.1, $t(p\phi) = t(p)(\sigma, \gamma)$ for some σ that satisfies this condition and γ such that Z' produces γ in response to σ after (s, g). Since $p\phi \in LR_{Abs(Z')}$, there exists $m' \in M$ such that $((m, \sigma), (\gamma, m')) \in \phi$. Furthermore, since Φ is observable, m' is unique. Since $(s\sigma, g\gamma) \in f_Z$, there exist $\phi' \in \Phi$ and $m'' \in M$ such that $p\phi' \in L_{A_Z}$ and $((m, \sigma), (\gamma, m'')) \in \phi'$. Since Φ is output-distinguishable, $\phi = \phi'$ and, hence, $m' = m''$. Thus the result follows. □

Conversely, the test process may also be used to establish that the dr-reachable sequences from the specification are also present in the language defined by the abstraction of the implementation. In particular, this result will be used later, (Lemma 7.1) to show that dr-distinguishable states in Z correspond to distinguishable states in $A_{Abs(Z')}$.

Lemma 6.3. Let us suppose that Φ is input-uniform, output-distinguishable and observable. Let $t : \Phi^* \longrightarrow \Sigma^* \times \Gamma^*$ be a test process of Z, $p, p' \in \Phi^*$ and let $(s, g) = t(p)$. Suppose $p \in LR_{Abs(Z')}$ and $t(pp') \in f_Z$. Then $t(p) \in f_Z$ and, by Lemma 6.2, there exists $q \in Q$ such that $F(q_0, p) = q$ and there exists unique $m \in M$ such that $((m_0, s), (g, m)) \in \|p\|$. If $p' \in LDR_Z(q, m)$ then $pp' \in L_{A_{Abs(Z')}}$.

Proof. Let $(s', g') = t(pp')$. We prove by induction on the length of p' that $pp' \in L_{A_{Abs(Z')}}$ and there exists unique $m' \in M$ such that $((m_0, s'), (g', m')) \in \|pp'\|$. The result clearly holds for the base case ϵ.

Let us suppose that the result holds for p' and let $\phi \in \Phi$ such that $p'\phi \in LDR_Z(q, m)$. Since Φ is input-uniform, there exists $\sigma \in \Sigma$ such that $(m', \sigma) \in dom\ \phi$. Then, by rule (2) of Definition 6.1, $t(pp'\phi) = t(pp')(\sigma, \gamma)$ for some σ that satisfies this condition and γ such that Z' produces γ in response to σ after (s', g'). Then there exist $\phi' \in \Phi$ and $m'' \in M$ such that $pp'\phi' \in L_{A_{Abs(Z')}}$ and $((m', \sigma), (\gamma, m'')) \in \phi'$. Since $(s'\sigma, g'\gamma) \in f_Z$, there exist $\phi'' \in \Phi$ and $m''' \in M$ such that $pp'\phi'' \in L_{A_Z}$ and $((m', \sigma), (\gamma, m''')) \in \phi''$. Since $p'\phi \in LDR_Z(q, m)$, $\phi = \phi''$. Since Φ is output-distinguishable, $\phi = \phi'$. Since Φ is observable, $m'' = m'''$. Thus the result follows. □

7. Test generation

The first step in the construction of the test data is the selection of two sets of sequences of processing relations, S_{dr} and W_{dr}, and of a relation d_{dr} on the states of Z as follows:

- $S_{dr} \subseteq LDR_Z$ is a finite set of d-realisable sequences such that
  - $\epsilon \in S_{dr}$ and
  - no state in Z is reached by more than one sequence in S_{dr} and so for every two distinct sequences $p_1, p_2 \in S_{dr}$, $F^*(q_0, p_1) \neq F^*(q_0, p_2)$.

  S_{dr} will be used to *reach* dr-reachable states in Z.
- $W_{dr} \subseteq \Phi^*$ is a finite set of processing relations. W_{dr} will be used to *dr-distinguish* between dr-distinguishable states of Z. W_{dr} is required to be non-empty, so when no sequences are used to dr-distinguish between states of Z, we will use $W_{dr} = \{\epsilon\}$ instead of $W_{dr} = \emptyset$.
- $d_{dr} : Q \longleftrightarrow Q$ is a relation on the states of Z that satisfies the following condition: for every two states $q_1, q_2 \in Q$, if $(q_1, q_2) \in d_{dr}$ then q_1 and q_2 are dr-distinguished by W_{dr}. The relation d_{dr} identifies the pairs of states that *are known* to be dr-distinguished by W_{dr}. For simplicity, d_{dr} is required to be symmetric.

It is desirable that

- S_{dr} is an dr-state cover of Z,
- W_{dr} is an dr-characterization set of Z and
- all pairwise dr-distinguishable states of Z are known to be dr-distinguished by W_{dr}, and so $(q_1, q_2) \in d_{dr}$ if and only if q_1 and q_2 are dr-distinguishable.

but these restrictions will not be introduced.

The set of all states of Z reached by sequences in S_{dr} is denoted by Q_{dr} and so $Q_{dr} = \{q \in Q \mid \exists p \in S_{dr}.F^*(q_0, p) = q\}$. As all sequences in S_{dr} are realisable, all states in Q_{dr} are dr-reachable. Furthermore, since $\epsilon \in S_{dr}$, the initial state of Z is contained in Q_{dr}.

Let $Q_1, \ldots Q_j$ denote the *maximal* sets of states of Z that are known to be pairwise dr-distinguished by W_{dr}; for all $1 \leq i \leq j$, if $q_1, q_2 \in Q_i$ and $q_1 \neq q_2$ then we have that $(q_1, q_2) \in d_{dr}$ and for every $q_3 \in Q \setminus Q_i$, there exists $q_1 \in Q_i$ such that $(q_1, q_3) \notin d_{dr}$. Let also $Q'_i = Q_i \cap Q_{dr}$, $1 \leq i \leq j$.

Consider Z in our example. As shown earlier, all states except q_3 are dr-reachable and all pairs of states are dr-distinguishable, $S_{dr} = \{\epsilon,\ enterNo\ enterNo\ correctPIN, enterNo\ enterNo\ correctPIN\ moveToCash\}$ is a dr-state cover of Z and $W_{dr} = \{moveToCash, withdrawCash, requestBalance\}$ is a dr-characterization set of Z.

Let us suppose that S_{dr} and W_{dr} are the chosen sets of sequences and, furthermore, all pairs of states are known to be dr-distinguished by W_{dr} and so $(q, q') \in d_r$ if and only if $q \neq q'$. Then there is one maximal set of states known to be pairwise dr-distinguished by $W_{dr} : Q_1 = Q$. Thus, in the example in Figure 1, $Q'_1 = Q_{dr} = \{q_0, q_1, q_2\}$.

Given a state $q \in Q_{dr}$, let $p_q \in S_{dr}$ denote the unique sequence in S_{dr} that reaches q; by the minimality of S_{dr}, p_q is well defined. Let us suppose that a test process $t : \Phi^* \longrightarrow \Sigma^* \times \Gamma^*$ has been defined for all sequences of processing relations in S_{dr}.

Given a state $q \in Q_{dr}$, the set $V(q)$ is defined to consist of all sequences $x \in \Phi^* \setminus \{\epsilon\}$ for which

- $p_q x \in LR_Z$,
- there exists i, $1 \leq i \leq j$, such that x visits states from Q_i exactly $n' - card(Q'_i) + 1$ times when followed from q in A_Z (the initial state of the path is not included in the count) and this condition does not hold for any proper prefix of x. More formally:
  - there exists i, $1 \leq i \leq j$, such that $card(\{y \mid y \in pref(x) \setminus \{\epsilon\} \wedge F^*(q, y) \in Q_i\}) = n' - card(Q'_i) + 1$ and
  - for all i, $1 \leq i \leq j$, and all $x_1 \in pref(x) \setminus \{x\}$, $card(\{y \mid y \in pref(x_1) \setminus \{\epsilon\} \wedge F^*(q, y) \in Q_i\}) < n' - card(Q'_i) + 1$.

The essential idea is that in order to determine whether $Fail$ is reachable we search for a minimal path to $Fail$. Informally, $V(q)$ is thus defined to contain only "minimal" paths of the Product Machine $A_{P(Z,Abs(Z'))}$ that may reach $Fail$. A minimal path must not have visited any pair of states ($(p, p') \in Q \times Q'$) twice and, furthermore, cannot contain pairs of states that have already been reached by the sequences in S_{dr}. Since we do not know the Product Machine we use a sufficient condition that ensures that at least one state of the Product Machine has been repeated. If a path x visits states from some Q_i, a tester can use W_{dr} after each prefix of x to distinguish between the corresponding states visited along x in Z'. Consequently, if states from Q_i are visited n_i times along a minimal path x, then for there to have been no repeated state of the Product Machine we must have visited n_i distinct states in Z'. Thus, n_i cannot exceed the upper bound n' on the number of states of Z' plus one (for the $Fail$ state). In addition, there are $card(Q'_i)$ states from Q_i that can be reached by sequences from S_{dr}. As S_{dr} will also reach the corresponding states of Z', this will leave $card(Q'_i)$ fewer pairs of states to explore. Thus, $n_i \leq n' - card(Q'_i) + 1$.

Given S_{dr}, W_{dr} and d_{dr}, the definition of $V(q)$ coincides with that given for a deterministic specification [Ipa06]. Note that, when Z is controllable, $LR_Z = L_{A_Z}$, so the definition reduces to that given in [HH04] for this case.

The set $V(q)$ can be constructed by devising a successor tree in which each path x from the root q corresponds to a realisable sequence $p_q x$. A path meets the termination criterion when it visits states from some Q_i exactly $n' - card(Q'_i) + 1$ times (some $1 \leq i \leq j$). In this case, the path need not be extended further and so the node is a leaf. A formal description of the procedure is given below. The procedure not only constructs $V(q)$, but also the values of a test process t for the sequences in $\{p_q\}V(q)$. It will transpire that these input/output sequences are used in testing.

In what follows, if Z' produces $g \in \Gamma^*$ in response to $s \in \Sigma^*$, $out_\sigma(s, g)$ will denote the output produced by Z' in response to the input σ after (s, g). Furthermore, given $m \in M$ and $\phi \in \Phi$ with $(m, \sigma) \in dom\ \phi$, $mem_\phi(m, \sigma, \gamma)$ will denote the memory value m' such that $((m, \sigma), (\gamma, m')) \in \phi$, if this exists; otherwise, $mem_\phi(m, \sigma, \gamma)$ will take a special value $\mu \notin M$, not contained in the original memory set. The definition of mem can be extended to take sequences of elements, giving $mem_p(m, s, g)$ for $p \in \Phi^*$, $s \in \Sigma^*$ and $g \in \Gamma^*$. For simplicity, in the following procedure it is assumed that $(s_q, g_q) = t(p_q)$, $m_q = mem_{p_q}(m_0, s_q, g_q)$ and the sets $Q_1, \ldots Q_j$ and $Q'_1, \ldots Q'_j$ have already been determined.

```
Input $Z$, $n'$, $q$, $p_q$, $(s_q, g_q)$, $m_q$, $Q_1, \ldots, Q_j$ and $card(Q'_1), \ldots, card(Q'_j)$;
$n_1 := 0, \ldots, n_j = 0$; $X := \emptyset$; $Y := \{((\epsilon, \epsilon, \epsilon), (q, m_q), (n_1, \ldots, n_j))\}$;
Repeat
  For $y$ in $Y$ do
    $Y := Y \setminus y$; $((p, s, g), (q, m), (n_1, \ldots, n_j)) := y$;
    If $m = \mu$ then $X = X \cup \{(p, s, g)\}$
    Else
      For $\phi$ in $\Phi$ such that $(q, \phi) \in dom\ F$ do
        Find $\sigma \in \Sigma$ such that $(m, \sigma) \in dom\ \phi$;
        If such $\sigma$ was found then
          $\gamma = out_\sigma(s, g)$;
          $m' = mem_\phi(m, \sigma, \gamma)$;
          For $i := 1$ to $j$ do
            If $F(q, \phi) \in Q_i$ then $n'_i := n_i + 1$
            Else $n'_i := n_i$;
          If there exists $i$, $1 \leq i \leq n$, such that $n'_i = n' - card(Q'_i) + 1$ then
            $X = X \cup \{(p\phi, s\sigma, g\gamma)\}$
            Else $Y = Y \cup \{((p\phi, s\sigma, g\gamma), (F(q, \phi), m')), (n'_1, \ldots, n'_j))\}$;
Until $Y = \emptyset$;
$V = \emptyset$; $TV = \emptyset$;
For $(x_1, x_2, x_3)$ in $X$ do
  $V = V \cup \{x_1\}$; $TV = TV \cup \{(p_q x_1, s_q x_2, g_q x_3)$;
Output $V$, $TV$.
```

Each iteration of the algorithm involves determining which elements of Y satisfy the termination criterion and thus do not need extending; these are transferred into X. The remaining elements are extended and the iteration continues. If the input/output pair produced by the test process has triggered a different processing relation in the IUT (there is no m' such that $((m, \sigma), (\gamma, m')) \in \phi)$), $m' = \mu$ and the sequence will be pruned. The algorithm outputs the set $V(q)$ (in which the sequences that are not triggered in the IUT are trimmed) and the values of a test process t for sequences in $\{p_q\}V(q)$.

As shown in [Ipa06], each sequence in $V(q)$ has length at most $n \cdot n'$. Thus $V(q)$ is finite and can be computed.

In our example all states are known to be dr-distinguished by W_{dr} so all the paths in the tree will have the same length, $n' - card(Q'_1) + 1 = n' - 2$. Thus, for any state q of Z, $\{p_q\}V(q) = \{p_q\}\Phi[n' - 2] \cap \mathrm{LR}_Z$.

Once we have constructed the sets $V(q)$, we define

$$U = \bigcup_{q \in Q_{dr}} \{p_q\}pref(V(q))$$

and

$$U_p = \bigcup_{q \in Q_{dr}} \{p_q\}(pref(V(q)) \setminus V(q))$$

.

Then a test process $t : \Phi^* \longrightarrow \Sigma^* \times \Gamma^*$ need only be applied to sequences in $UW_{dr} \cup U_p\Phi$.

The remainder of the section validates this construction.

Lemma 7.1. Let $p_1, p_2 \in LR_Z$, $q_1, q_2 \in Q$ such that $F^*(q_0, p_1) = q_1$ and $F^*(q_0, p_2) = q_2$ and $q'_1, q'_2 \in Q'$ such that $F'^*_{abs}(q'_0, p_1) = q'_1$ and $F'^*_{abs}(q'_0, p_2) = q'_2$. Let us suppose that W_{dr} dr-distinguishes between q_1 and q_2 in Z. If for all $p \in \{p_1, p_2\}W_{dr}$, $t(p) \in f_Z$ then W_{dr} distinguishes between q'_1 and q'_2 in $A_{Abs(Z')}$.

Proof. Let $m_1, m_2 \in M$, $g_1, g_2 \in \Gamma^*$, $s_1, s_2 \in \Sigma^*$, such that $t(p_1) = (s_1, g_1)$, $(g_1, m_1) \in \|p_1\|(m_0, s_1)$, $t(p_2) = (s_2, g_2)$, and $(g_2, m_2) \in \|p_2\|(m_0, s_2)$. Since W_{dr} dr-distinguishes between q_1 and q_2 in Z, there exists $x \in W_r$ such that $x \in LDR_Z(q_1, m_1) \setminus LR_Z(q_2, m_2)$ or $x \in LDR_Z(q_2, m_2) \setminus LR_Z(q_1, m_1)$. Without loss of generality, assume $x \in LDR_Z(q_1, m_1) \setminus LR_Z(q_2, m_2)$. Since $t(p_1 x) \in f_Z$, by Lemma 6.3, $x \in L_{A_{Abs(Z')}}(q'_1)$.

On the other hand, since $t(p_2x) \in f_Z$, by Lemma 6.2 $x \notin L_{A_{Abs(Z')}}(q'_2)$. Thus W_r distinguishes between q'_1 and q'_2 in $A_{Abs(Z')}$. $\square$

Lemma 7.2. Let $q \in Q$ and $x \in V(q)$ with $p_qx \in L_{A_{Abs(Z')}}$. If for all $p \in S_{dr}W_{dr} \cup \{p_q\}pref(x)W_{dr}$, $t(p) \in f_Z$ then the path in $A_{P(Z,Abs(Z'))}$ formed by following x after p_q either contains a loop or meets a state, other than the root state, that has already been reached by some sequence in S_{dr}.

Proof. For simplicity, in what follows we will use $path_Z(x, p_q)$, $path_{Z'}(x, p_q)$ and $path_{Z,Z'}(x, p_q)$ to denote the paths formed by following x after p_q in A_Z, $A_{Abs(Z')}$ and $A_{P(Z,Abs(Z'))}$, respectively. The root states are not included when referring to these paths. First note that, by Lemma 5.1, path $path_{Z,Z'}(x, p_q)$ exists in $\mathrm{A}_{P(Z,Abs(Z'))}$.

We prove the lemma by contradiction. Assume $path_{Z,Z'}(x, p_q)$ is cycle-free and does not meet any state reached by sequences in S_r, other than the root state. Let i be such that $path_Z(x, p_q)$ visits states from Q_i exactly $n' - card(Q'_i) + 1$ times. By Lemma 7.1, since W_{dr} pairwise dr-distinguishes between the states in Q_i, it also pairwise distinguishes between the corresponding states in $A_{Abs(Z')}$. Thus, since $path_{Z,Z'}(x, p_q)$ is cycle-free, $path_{Z'}(x, p_q)$ visits at least $n' - card(Q'_i) + 1$ distinct states of $Abs(Z')$ and the sequences in S_r will reach at least another $card(Q'_i)$ states of $Abs(Z')$. This implies that Z' has more than n' states, providing a contradiction as required. $\square$

Lemma 7.3. Let us suppose that the $Fail$ state of $A_{P(Z,Abs(Z'))}$ is reachable. If for all $p \in UW_{dr}$, $t(p) \in f_Z$ then $Fail$ can be reached by some sequence from $U_p\Phi$.

Proof. Let us suppose that $Fail$ is reachable. Then there exist $p_1 \in L_{A_Z} \cap L_{A_{Abs(Z')}}$, $\phi \in \Phi$ and $(q_1, q'_1) \in Q \times Q'$ such that p_1 reaches (q_1, q'_1) in $A_{P(Z,Abs(Z'))}$, $(q_1, \phi) \notin dom\ F$ and $(q'_1, \phi) \in dom\ F'_{abs}$. Since $\epsilon \in S_r$, $p_1 \in S_r\Phi^*$. Let $i \geq 0$ be the minimum integer for which there exists a sequence in $S_r\Phi^i$ that reaches (q_1, q'_1) in $A_{P(Z,Abs(Z'))}$. Let $p_2 = p_qx$ be such a sequence, $p_q \in S_r$, $x \in \Phi^i$. By Lemma 5.1, $p_2 \in L_{A_Z} \cap L_{A_{Abs(Z')}}$. Thus, since $Abs(Z')$ is controllable, $p_2 \in LR_Z$. Then, either p_2 is contained in $pref(V(q))$ or extends some sequence from $V(q)$, i.e. $x \in V(q)\Phi^*$. Since i is the minimum integer with the above property, the path in $A_{P(Z,Abs(Z'))}$ formed by following x after p_q will be cycle-free and will not meet any state reached by sequences in S_r. Then, by Lemma 7.2, $x \in pref(V(q)) \setminus V(q)$. Thus $x\phi \in (pref(V(q)) \setminus V(q))\Phi$, so $p_2\phi \in U_p\Phi$. Since p_2 reaches (q_1, q'_1), $p_2\phi$ will reach $Fail$ in $A_{P(Z,Abs(Z'))}$. Thus the result follows. $\square$

Lemma 7.4. If for all $p \in UW_{dr} \cup U_p\Phi$, $t(p) \in f_Z$ then the $Fail$ state of $A_{P(Z,Abs(Z'))}$ is not reachable.

Proof. We provide a proof by contradiction. Assume $Fail$ is reachable. Then, by Lemma 7.3, $Fail$ can be reached by some sequence p from $U_p\Phi$. By Lemma 5.2, $p \in L_{A_{Abs(Z')}} \setminus L_{A_Z}$. On the other hand, since $t(p) \in f_Z$, by Lemma 6.2, $p \in L_{A_Z}$. This provides a contradiction, as required. $\square$

Theorem 7.1. The $Fail$ state of $A_{P(Z,Abs(Z'))}$ is not reachable if and only if for all $p \in UW_{dr} \cup U_p\Phi$, $t(p) \in f_Z$.

Proof. "$\Leftarrow$": Follows from Lemma 7.4.

"$\Rightarrow$" By Lemma 6.1, for every $p \in \Phi^*$, $t(p) \in f_{Z'}$. Then the result follows from Lemma 5.5. $\square$

Theorem 7.2. Z' conforms to Z if and only if for all $p \in UW_{dr} \cup U_p\Phi$, $t(p) \in f_Z$.

Proof. Follows from Theorem 7.1 and Lemma 5.5. $\square$

Note that if all the states of Z are dr-reachable and pairwise dr-distinguishable, S_{dr} is a dr-state cover of Z, W_{dr} is a dr-characterization set of Z and all states of Z are known to be pairwise dr-distinguished by W_{dr}, then

$$U = S_{dr}\Phi[n' - n + 1] \cap \mathrm{LR}_Z$$

where n represents the number of states of Z, so the method reduces to an extension of the W-method [Cho78] to SXMs. This particular case is a generalization of the result given in [IH97], which extends the W-method only to *controllable* deterministic SXM specifications.

On the other extreme, if $S_{dr} = \{\epsilon\}$ and $W_{dr} = \{\epsilon\}$ then $U = \Phi[n'n]$. In most practical applications, however, the state counting approach will produce far fewer test sequences.

We have seen that in our example, for any state q of Z, $\{p_q\}V(q) = \{p_q\}\Phi[n'-2] \cap \mathrm{LR}_Z$. Recall also that $S_{dr} = \{\epsilon, enterNo\ enterNo\ correctPIN, enterNo\ enterNo\ correctPIN\ moveToCash\}$ and so we get:

$$U = \{\epsilon, enterNo\ enterNo\ correctPIN, enterNo\ enterNo\ correctPIN\ moveToCash\}\Phi[n'-2] \cap \mathrm{LR}_Z$$

We thus simply apply a test process to all $p \in UW_{dr} \cup U_p\Phi$, for this value of U and $W_{dr} = \{moveToCash, withdrawCash, requestBalance\}$.

8. Complexity

We now examine the size of the generated test suite and the complexity of the test generation algorithm. For $U = S_{dr}\Phi[n'-n+1]$, as given by the application of the W-method to the associated finite automaton, the number of sequences in UW_{dr} is at most $n^2 \cdot k^{n'-n+1}$ and the total length of all sequences in UW_{dr} is at most $n^2 \cdot n' \cdot k^{n'-n+1}$, where $k = card(\Phi)$ [Cho78]. Typically, only a small fraction of the sequences in $S_{dr}\Phi[n'-n+1]$ are realisable, so the actual size of test suite is significantly lower. In the worst case, when $S_{dr} = W_{dr} = \{\epsilon\}$, the upper bounds are proportional to $k^{n' \cdot n}$. However, this extreme is not normally encountered in practice. In usual applications, most states will be dr-reachable and pairwise dr-distinguishable. When n' is considerably larger than n, additional criterion can be used to prune the sequences in U, as discussed in [Ipa06].

As each step of the test generation algorithm selects an input symbol and computes the next memory value, the complexity of this algorithm will be proportional to the total length of all sequences in U, the number of input symbols and the effort required to compute the new memory. Thus, for the case in which all the states of Z are dr-reachable and pairwise dr-distinguishable, S_{dr} is an dr-state cover, W_{dr} is an dr-characterization set of Z and all states of Z are known to be pairwise dr-distinguished by W_{dr}, the complexity will be no more than $C \cdot r \cdot n^2 \cdot n' \cdot k^{n'-n+1}$, where $r = card(\Sigma)$ and C is the maximum effort needed by a processing relation to compute the next memory value, given the input and the current memory. Note that none of the above depend on the size of the input alphabet and instead depend on the size of Φ. This is because we abstract away from the input values and consider relations in Φ.

9. Conclusions

Stream X-machines (SXMs) are a type of extended finite state machine that can be used for system specification. Associated with SXMs is an approach to development in which a system is built from trusted components. One of the great benefits of this approach is that it is possible to produce a finite test suite that determines correctness as long as certain properties hold.

Traditionally, work on using SXMs in development had two major limitations. First, it considered only deterministic SXMs. Recent work has extended the approach to non-deterministic specifications, an important generalization since non-determinism aids abstract and is highly appropriate for specifications. Second, the work on testing from SXMs has included the condition that the specification is controllable: all paths through the specification SXM are feasible. Unfortunately, many specifications are not controllable. This paper is the first to show how the controllability property can be weakened for non-deterministic specifications. The paper also includes an algorithm that produces a finite test suite, that is guaranteed to determine correctness subject to certain conditions holding, from a non-controllable non-deterministic SXM.

Testing is relative to a fault domain which contains deterministic SXMs that satisfy certain test hypotheses: testing with the algorithm given in this paper is guaranteed to produce a failure if the implementation is faulty and is a member of the fault domain. As usual, the fault domain places an upper bound on the number of states of the implementation. It also assumes that every function used in the implementation conforms to some relation contained in the specification: the implementation has been developed using trusted components. While we do not assume that the specification is controllable we do assume that the implementation is controllable. However, since in practice the memory is finite, there is always a controllable stream X-machine that models the IUT.

It has previously been shown how design for test conditions required for generating test suites from a SXM specification can be weakened by incorporating a test environment that restricts the inputs used in testing (see, for example, [HH04, IH00]). It should be straightforward to include a test environment in the results given in this paper. There should also be scope for applying an adaptive approach to test generation in which test generation is informed by the input/output sequences that have been observed in testing (see, for example, [CLL04, Hie04, TN92, YKK97]). Finally, it should be possible to generalize the test generation algorithm to work with non-deterministic implementations.

References

[ABC$^+$02] Joaquin Aguado, Tudor Balanescu, Anthony J. Cowling, Marian Gheorghe, Mike Holcombe, and Florentin Ipate. P systems with replicated rewriting and stream x-machines (eilenberg machines). *Fundam. Inform.*, 49(1-3):17–33, 2002.

[BGGV99] T. Balanescu, H. Georgescu, M. Gheorghe, and C. Vertan. Communicating stream X-machines systems are no more than X-machines. *Journal of Universal Computing*, 5(9):494–507, 1999.

[BGH03] Francesco Bernardini, Marian Gheorghe, and Mike Holcombe. P x systems = p systems + x machines. *Natural Computing*, 2(3):201–213, 2003.

[BH01] K. Bogdanov and M. Holcombe. Statechart testing method for aircraft control systems. *The Journal of Software Testing, Verification and Reliability*, 11(1):39–54, 2001.

[BHI$^+$06] K. Bogdanov, M. Holcombe, F. Ipate, L.Seed, and S. Vanak. Testing methods for x-machines, a review. *Formal Aspects of Computing*, 18(1):3–30, 2006.

[CGV00] Anthony J. Cowling, Horia Georgescu, and Cristina Vertan. A structured way to use channels for communication in x-machine systems. *Formal Asp. Comput.*, 12(6):485–500, 2000.

[Cho78] T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[CLL04] K.-Y. Cai, Y.-C. Li, and K. Liu. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology*, 46(15):989–1000, 2004.

[DSA$^+$99] R. Dssouli, K. Saleh, E. Aboulhamid, E. En-Nouaary, and C. Bourhfir. Test development for communications protocols: towards automation. *Computer Networks*, 31:1835–1872, 1999.

[DU04] A. Y. Duale and M. U. Uyar. A method enabling feasible conformance test sequence generation for EFSM models. *IEEE Transactions on Computers*, 53(5):614–627, 2004.

[Eil74] S. Eilenberg. *Automata, languages and machines*, volume A. Academic Press, 1974.

[FUDA03] M. A. Fecko, M. U. Uyar, A. Y. Duale, and P. D. Amer. A technique to generate feasible tests for communications systems with multiple timers. *IEEE/ACM Transactions on Networking*, 11:796–809, 2003.

[HH00] R. M. Hierons and M. Harman. Testing conformance to a quasi-non-deterministic stream x-machine. *Formal Aspects of Computing*, 12(6):423–442, 2000.

[HH04] R. M. Hierons and M. Harman. Testing conformance of a deterministic implementation against a non-deterministic stream x-machine. *Theoretical Computer Science*, 323(1–3):191–233, 2004.

[HI98] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*. Springer-Verlag, 1998.

[Hie03] R. M. Hierons. Generating candidates when testing a deterministic implementation against a non-deterministic finite-state machine. *The Computer Journal*, 46(3):307–318, 2003.

[Hie04] R. M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Transations on Computers*, 53(10):1330–1342, 2004.

[Hol93] M. Holcombe. An integrated methodology for the specification, verification and testing of systems. *The Journal of Software Testing, Verification and Reliability*, 3(3/4):149–163, 1993.

[HU06] R. M. Hierons and H. Ural. Optimizing the length of checking sequences. *IEEE Transactions on Computers*, 55(5):618–629, 2006.

[IH97] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63:159–178, 1997.

[IH00] F. Ipate and M. Holcombe. Generating test sets from non-deterministic stream X-machines. *Formal Aspects of Computing*, 12(6):443–458, 2000.

[Ipa04] F. Ipate. Complete deterministic stream x-machine testing. *Formal Aspects of Computing*, 16(4):374–386, 2004.

[Ipa06] F. Ipate. Testing against a non-controllable stream x-machine using state counting. *Theoretical Computer Science*, 353(1–3):291–316, 2006.

[IT97] ITU-T. *Recommendation Z.500 Framework on formal methods in conformance testing.* International Telecommunications Union, Geneva, Switzerland, 1997.

[JHR04] D. Jackson, M. Holcombe, and F. Ratnieks. Trail geometry gives polarity to ant foraging networks. *Nature*, 432:907–909, 2004.

[KEK03] Petros Kefalas, George Eleftherakis, and Evangelos Kehris. Communicating x-machines: a practical approach for formal and modular specification of large systems. *Information & Software Technology*, 45(5):269–280, 2003.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite–state machines – a survey. *Proceedings of the IEEE*, 84(8):1089–1123, 1996.

[PBG04] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Transactions on Software Engineering*, 30(1):29–42, 2004.

[PY05a] A. Petrenko and N. Yevtushenko. Testing from partial deterministic FSM specifications. *IEEE Transaxtions on Computers*, 54(9):1154–1165, 2005.

[PY05b] Alexandre Petrenko and Nina Yevtushenko. Conformance tests as checking experiments for partial nondeterministic FSM. volume 3997 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2005.

[PYB96] A. Petrenko, N. Yevtushenko, and G.v. Bochmann. Testing deterministic implementations from nondeterministic fsm specifications. In *Proc. of 9th International Workshop on Testing of Communicating Systems (IWTCS'96*, pages 125–140, 1996.

[RHRT05] Christopher A. Rouff, Michael G. Hinchey, James L. Rash, and Walter F. Truszkowski. Towards a hybrid formal method for swarm-based exploration missions. In *SEW*, pages 253–264, 2005.

[RMN06] I. Rodríguez, M. G. Merayo, and M. Núñez. A logic for assessing sets of heterogeneous testing hypotheses. In *18th IFIP TC6/WG6.1 International Conference on the Testing of Communicating Systems (TestCom 2006)*, pages 39–54, 2006.

[SGK04] Ioanna Stamatopoulou, Marian Gheorghe, and Petros Kefalas. Modelling dynamic organization of biology-inspired multi-agent systems with communicating x-machines and population p systems. In *Workshop on Membrane Computing*, pages 389–403, 2004.

[SH06] Rod H. Smallwood and Mike Holcombe. The epitheliome project: multiscale agent-based modeling of epithelial cells. In *ISBI*, pages 816–819, 2006.

[TN92] P. Tripathy and K. Naik. Generation of adaptive test cases from non-deterministic finite state models. In *Proceedings of the 5th International Workshop on Protocol Test Systems*, pages 309–320, Montreal, September 1992.

[UW03] H. Ural and D. Whittier. Distributed testing without encountering controllability and observability problems. *Information Processing Letters*, 88(3):133–141, 2003.

[YKK97] S. Yoo, M. Kim, and D. Kang. An approach to dynamic protocol testing. In *IFIP TC6 10th International Workshop on Testing of Communicating Systems*, pages 183–199, Cheju Island, Korea, September 1997. Chapman and Hall, London.