

# Testing against a non-controllable stream X-machine using state counting

Florentin Ipate\*

*Department of Computer Science and Mathematics, University of Pitesti, Str Targu din Vale 1, 0300 Pitesti, Romania*

Received 23 June 2005; received in revised form 6 December 2005; accepted 13 December 2005

Communicated by D. Sannella

---

## Abstract

*Stream X-machines* are a form of extended finite state machines that has received extensive study in recent years. A stream X-machine describes a system as a finite set of states, an internal store, called memory, and a finite number of transitions between the states, labelled by function names (the processing functions). One of the great benefits of using a stream X-machine to specify a system is its associated *testing method*. Under certain *design for test conditions*, this method produces a test suite that can determine the *correctness* of the implementation, provided that the processing functions of the stream X-machine specification have been correctly implemented (this can be checked by a separate testing process, using the same method or alternative functional methods). However, the application of the stream X-machine based testing method is often encumbered by the restrictive design for test conditions required. In practical applications, these conditions are achieved by designing extra functionality that will have to be disabled after testing has been completed. This is a time consuming process and can often be a source of error. This paper provides a strong generalisation of the existing method, which requires much laxer design for test conditions; these are naturally satisfied in practical applications and, furthermore, can be introduced into any stream X-machine specification without the need to add extra functionality. Consequently, the generalised method can be applied to virtually any system that can be specified by a stream X-machine.

© 2006 Elsevier B.V. All rights reserved.

*Keywords:* Specification based testing; Test generation; Formal specifications; Stream X-machines; Finite state machines

---

## 1. Introduction

Formal methods have been regarded by many researchers as a solution to the problem of building high quality software. The use of formal specifications and models in software development eliminates the opportunity for ambiguity and allows the application of, possibly automated, formal analysis.

One approach to formally specifying a system is to use a form of extended finite state machine called a *stream X-machine* [28,32]. A stream X-machine is a type of X-machine [17,27,28] that describes a system as a finite set of states, an internal store, called memory, and a number of transitions between the states. A transition is triggered by an input value, produces an output value and may alter the memory. A stream X-machine may be modelled by a

---

\* Tel.: +40 21 4108964; fax: +40 248 216448.

E-mail address: [fipate@ifsoft.ro](mailto:fipate@ifsoft.ro).

finite automaton (the *associated finite automaton*) in which the arcs are labelled by function names (the *processing functions*). Stream X-machines combine the dynamic features of finite state machines with data structures, thus sharing the benefits of both these worlds. Various case studies [28,18,40] have demonstrated the value of the stream X-machine as a specification method, especially for interactive systems. A tool for writing stream X-machine specifications has also been constructed [39].

The X-machine model has also been studied from a theoretical point of view: various subclasses have been defined [32,4,5,23], the minimality issue has been investigated [30] and a *refinement* of stream X-machines [34,37] has been formalised. Furthermore, several models of *communicating stream X-machines* have been devised and their applicability to real applications has been demonstrated [9,3,15,22,38]. Communicating stream X-machines have also been used for simulating and verifying P-systems [1,6,10].

However, even where a formal specification or model is used, it is important to test the implementation [19]; this is the product we are ultimately interested in. While the presence of a formal model or specification of the required behaviour may allow test generation to be automated [21,24,16,41,13] it is still often difficult to deduce much from the *implementation under test* (IUT) behaving correctly on the test suite produced. Ideally, we should produce tests that are capable of providing a high degree of confidence in the correctness of the IUT while simplifying the problem of test generation to allow automation.

One of the great benefits of using a stream X-machine to specify a system is that, under certain well defined conditions, it is possible to produce a test suite that can determine the correctness of the IUT [33,28]. These conditions fall into two categories: *design for test conditions*, which place restrictions on the specification, and *test hypotheses*, which place restrictions on the IUT. The design for test conditions are rules that, if followed in the design process, will produce a system that is more easily testable. The test hypotheses require the system to be made of *correct* components, in other words the processing functions of the stream X-machine specification are required to be correctly implemented. In practice, this is checked by a separate process [28,35]: depending on the nature of the function, it can be tested using the same method or alternative functional methods, for example category partition testing [43] or a variant. Furthermore, recent results enable the testing of the processing functions to be integrated into the testing of the overall system [31]. A direct consequence of the test hypotheses is that, unlike other extended finite state machine based approaches [12,41], the test generation 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. The stream X-machine based testing method was first developed for deterministic stream X-machines (those machines where, in any state and for any memory value, an input triggers at most one transition) [33,28] and was later extended to non-deterministic stream X-machines [36] and communicating stream X-machines [38]. In the case of non-deterministic specifications, both equivalence [36] and conformance [25,26] have been used as notion of correctness. The effectiveness of the method has been validated by many industrial case studies [11].

However, the application of the stream X-machine based testing method is often encumbered by the strictness of the design for test conditions required. There are two such conditions: *output-distinguishability* and *controllability*. The first requires that every processing function can be distinguished by examining the output produced when an input is applied to any given memory value. Controllability basically means that every path in the associated automaton can actually be driven by suitable input sequences. Whilst the first condition is quite natural and can be satisfied by a suitable enrichment of the observed output, controllability is seldom met by non-trivial specifications. In practical applications, controllability is enforced on a stream X-machine specification by designing extra input symbols that are not used in normal function and will have to be disabled after testing has been completed. This is a time consuming process and can often be a source of error.

This paper generalises the existing stream X-machine based testing method by considering specifications that do not meet the controllability requirement. This is replaced by a much laxer condition, called *input-uniformity*, which basically requires all memory values that are produced by the application of any single sequence of processing functions to any single memory to be processed in a uniform way by any processing function—that is, any function can either process all such memory values or none. The relaxed condition is naturally satisfied in practical applications of the method [28,11] and, furthermore, it can be achieved through a suitable refinement of the processing functions, without the need to design extra functionality. Consequently, the generalised method can be applied to virtually any stream X-machine specification.

As a consequence of removing the controllability requirement from the specification, the test generation strategy will have to be changed. When the specification is controllable, all paths can actually be driven by input sequences,

so the test generation procedure will basically be a generalisation of Chow's  $W$ -method for deterministic finite state machines [33]. When the specification is not controllable, the  $W$ -method can no longer be applied. Instead, a *state-counting* approach will be used, which involves the construction of a product machine of the specification and the implementation. State-counting was originally used for conformance testing of a deterministic implementation against a non-deterministic finite state machine [44] and has been recently applied to testing of a deterministic implementation against a *controllable* non-deterministic stream X-machine specification [26].

The paper is structured as follows. Sections 2 and 4 introduce basic concepts of finite state machines and stream X-machines, respectively, while Section 3 presents Chow's  $W$ -method for testing against a deterministic finite state machine specification. The design for test conditions, both in their original form (for controllable stream X-machines) and in the new, relaxed, form are given in Section 5. Section 6 discusses the issue of reaching and identifying the states of (possibly) non-controllable stream X-machines. The following five sections are dedicated to the generalised stream X-machine based testing method: Section 7 states the pre-requisites of the method; some preliminary results are given in Section 8; Section 9 defines the product machine used in state-counting, while Section 10 defines a test function as a means of converting sequences of processing functions derived from the product machine into input sequences; finally, the construction of the test suite and the results that validate this construction are given in Section 11. Section 12 discusses the complexity of the method, while the next section provides a technique for reducing the size of large test suites. Conclusions are drawn and further work is outlined in Section 14.

Before continuing, we introduce the notation used in the paper. For a finite alphabet  $A$ ,  $A^*$  denotes the set of all finite sequences with members in  $A$ .  $\varepsilon$  denotes the empty sequence. For a sequence  $a \in A^*$ ,  $|a|$  denotes the number of elements of  $a$  (in particular  $|\varepsilon| = 0$ ). For  $a, b \in A^*$ ,  $ab$  denotes the concatenation of sequences  $a$  and  $b$ .  $a^n$  is defined by  $a^0 = \varepsilon$  and  $a^n = a^{n-1}a$ ,  $n \geq 1$ . For  $U, V \subseteq A^*$ ,  $UV = \{ab | a \in U, b \in V\}$ ;  $U^n$  is defined by  $U^0 = \{\varepsilon\}$  and  $U^n = U^{n-1}U$ ,  $n \geq 1$ . Furthermore,  $U[n] = \bigcup_{0 \leq i \leq n} U^i$ . For a sequence  $a \in A^*$ ,  $b \in A^*$  is said to be a *prefix* of  $a$  if there exists a sequence  $c \in A^*$  such that  $a = bc$ . The set of all prefixes of  $a$  is denoted by  $pref(a)$ . For  $U \subseteq A^*$ ,  $pref(U) = \bigcup_{a \in U} pref(a)$ . For a relation or a (partial) function  $f : A \rightarrow B$ ,  $dom f$  denotes the domain of  $f$ . For a finite set  $A$ ,  $card(A)$  denotes the number of elements in  $A$ .

## 2. Deterministic finite state machines

This section introduces the deterministic finite state machine and related concepts and results that will be used later in the paper.

**Definition 2.1.** A *deterministic finite state machine (DFSM)*  $A$  is a tuple  $(\Sigma, \Gamma, Q, h, q_0)$ , as follows:

- $\Sigma$  is the finite *input alphabet*.
- $\Gamma$  is the finite *output alphabet*.
- $Q$  is the finite *set of states*.
- $h$  is the (partial) *next-state and output function*,  $h : Q \times \Sigma \rightarrow Q \times \Gamma$ ;  $h$  is usually described by a state-transition diagram.
- $q_0 \in Q$  is the *initial state*.

**Definition 2.2.**  $A$  is said to be *completely specified* if  $h$  is a total function. Otherwise  $A$  is said to be *partially specified*.

**Definition 2.3.** The (partial) function  $h : Q \times \Sigma \rightarrow Q \times \Gamma$  breaks up into two (partial) functions

- $h_1 : Q \times \Sigma \rightarrow Q$ ,
- $h_2 : Q \times \Sigma \rightarrow \Gamma$

having a common domain.  $h_1$  is called the *next-state function* and  $h_2$  the *output function*.

**Definition 2.4.** The next-state function  $h_1$  can be extended to a (partial) function  $h_1^* : Q \times \Sigma^* \rightarrow Q$  defined by

- $h_1^*(q, \varepsilon) = q, q \in Q$ ,
- $h_1^*(q, s\sigma) = h_1(h_1^*(q, s), \sigma), q \in Q, s \in \Sigma^*, \sigma \in \Sigma$ .

The output function  $h_2$  can be extended to a (partial) function  $h_2^* : Q \times \Sigma^* \rightarrow \Gamma^*$  defined by

- $h_2^*(q, \varepsilon) = \varepsilon, q \in Q,$
- $h_2^*(q, s\sigma) = h_2^*(q, s)h_2(h_1^*(q, s), \sigma), q \in Q, s \in \Sigma^*, \sigma \in \Sigma.$

**Definition 2.5.** Given  $q \in Q$ , the (partial) function computed by  $A$  in  $q$ , denoted by  $f_A^q$ , is defined by

$$f_A^q(s) = h_2^*(q, s), \quad s \in \Sigma^*.$$

The function computed by  $A$  in  $q_0$  is simply called the function computed by  $A$  and is denoted by  $f_A$ .

**Definition 2.6.** A state  $q \in Q$  is called *reachable* if there exists  $s \in \Sigma^*$  such that  $h_1^*(q_0, s) = q$ .  $A$  is called *reachable* if all states of  $A$  are reachable.

**Definition 2.7.** Given  $Y \subseteq \Sigma^*$ , two states  $q_1, q_2 \in Q$  are called *Y-equivalent* if for all  $s \in Y$ ,  $h_2^*(q_1, s) = h_2^*(q_2, s)$ . Otherwise  $q_1$  and  $q_2$  are called *Y-distinguishable*. If  $Y = \Sigma^*$  then  $q_1$  and  $q_2$  are simply called *equivalent* or *distinguishable*, respectively. Two DFSMs are called *(Y-)equivalent* or *(Y-)distinguishable* if their initial states are *(Y-)equivalent* or *(Y-)distinguishable*, respectively.

**Definition 2.8.**  $A$  is called *reduced* if every two distinct states of  $A$  are distinguishable.

**Definition 2.9.**  $A$  is called *minimal* if any DFSM that computes  $f_A$  has at least the same number of states as  $A$ .

**Theorem 2.1.**  $A$  is minimal if and only if  $A$  is reachable and reduced.

This is a well known result, for a proof see for example [17].

**Definition 2.10.** Let  $A = (\Sigma, \Gamma, Q, h, q_0)$  and  $A' = (\Sigma, \Gamma, Q', h', q'_0)$  be two DFSMs over the same input alphabet. Then a function  $g : Q \rightarrow Q'$  is called an *isomorphism* if

- $g$  is bijective,
- $g(q_0) = q'_0,$
- $g(h_1(q, \sigma)) = h'_1(g(q), \sigma), q \in Q, \sigma \in \Sigma,$
- $h_2(q, \sigma) = h'_2(g(q), \sigma), q \in Q, \sigma \in \Sigma.$

**Theorem 2.2.** For two minimal DFSMs  $A$  and  $A'$ ,  $f_A = f_{A'}$  if and only if  $A$  and  $A'$  are isomorphic.

This is a well known result, for a proof see for example [17]. Techniques for constructing the minimal DFSM that computes the same function as a given DFSM also exist, for more detail see for example [17,14].

One special case of DFSM used in this paper is that in which the input and output alphabets coincide ( $\Sigma = \Gamma$ ) and the output produced by  $h_2$  is always identical to the input. Such a machine is completely described by a tuple  $A = (\Sigma, Q, h_1, q_0)$  and in this paper it will be referred to as a *finite automaton* (FA)—the deterministic nature of the automaton will not be explicitly stated as any non-deterministic FA can be converted into an equivalent deterministic FA [14]. Since in this case the outputs are always identical to the inputs, the function computed by  $A$  (in the state  $q$ ) will be completely determined by its domain. This will be called the *language accepted* by  $A$  (in  $q$ ) and will be denoted by  $L_A$  (or  $L_A(q)$ ).

### 3. The $W$ method for DFSMs

We now turn our attention to DFSM based testing and, in particular, to the generation of test suites from a DFSM specification. Given a DFSM specification  $A$  and a set  $C$  of DFSMs, called the *fault domain*, a test suite is a finite set of input sequences which, if they produce the specified results when applied to the implementation, will establish that the implementation under test is functionally equivalent to the specification, provided that it is known that this implementation can be modelled by some DFSM  $A'$  in  $C$ .

Naturally, the fault domain  $C$  is identified by the assumptions one can make about the implementation. In principle, no information is available about the implementation, but in this case a test suite may not exist for even very simple DFSM specifications. There are a number of more or less realistic assumptions that one can make about the form and size of the implementation model  $A'$  and these, in turn, give rise to different techniques for generating test suites [41]. One of the least restrictive assumptions refers to the number of states in  $A'$  and is the basis for the *W-method* [13]: the difference between the number of states of the implementation model and that of the specification has to be at most  $k$ , a non-negative integer estimated by the tester.

The *W-method* involves the selection of two sets of input sequences, a state cover and a characterisation set, as defined next.

**Definition 3.1.**  $S \subseteq \Sigma^*$  is called a *state cover* of  $A = (\Sigma, \Gamma, Q, h, q_0)$  if  $\varepsilon \in S$  and for every state  $q \in Q$  there exists  $s \in S$  such that  $h_1^*(q_0, s) = q$ .

**Definition 3.2.**  $W \subseteq \Sigma^*$  is called a *characterisation set* of  $A = (\Sigma, \Gamma, Q, h, q_0)$  if any two distinct states of  $A$  are *W-distinguishable*.

Note that a state cover and a characterisation set exist if  $A$  is minimal. These concepts are illustrated by Example 3.1.

Then the test suite generated by the *W-method* is

$$U_k = S\Sigma[k + 1]W = S \left( \bigcup_{0 \leq i \leq k+1} \Sigma^i \right) W,$$

where

- $S$  is a state cover of the specification  $A$ .
- $W$  is a characterisation set of the specification  $A$ .

The idea is that the set  $S\Sigma[1]$  (usually called a *transition cover* of  $A$ ) ensures that all the states and all the transitions in  $A$  are also present in  $A'$  and  $\Sigma[k]W$  ensures that  $A'$  is in the same state as  $A$  after performing each transition. Note that the latter set contains  $W$  and also all sets  $\Sigma^i W$ ,  $1 \leq i \leq k$ . This ensures that  $A'$  does not contain extra states. If there were up to  $k$  extra states, then each of them would be reached by some input sequence of up to length  $k$  from the existing states.

**Theorem 3.1** (Chow [13]). *Let  $A = (\Sigma, \Gamma, Q, h, q_0)$  and  $A' = (\Sigma, \Gamma, Q', h', q'_0)$  be completely specified DFSMs,  $A$  minimal, such that  $\text{card}(Q') - \text{card}(Q) \leq k, k \geq 0$ . Then  $A$  and  $A'$  are equivalent if and only if  $A$  and  $A'$  are  $U_k$ -equivalent.*

The *W-method* was originally devised by Chow [13] for the case where the specification and the model of the implementation are both *completely specified* DFSMs. On the other hand, finite automata, which are used in the description of stream X-machines, are almost always partially specified. Thus, given the purpose of this paper, of particular interest is the application of the *W-method* to (possibly) partially specified machines. As shown by the following counterexample, the *W-method* in the above form does not work for partially specified DFSMs.

**Example 3.1.** Let us consider  $A = (\{a, b\}, \{a, b\}, \{0, 1\}, h, 0)$ , and  $A' = (\{a, b\}, \{a, b\}, \{0, 1\}, h', 0)$ , where  $h(0, a) = (1, a)$ ,  $h'(0, a) = (1, a)$ ,  $h'(0, b) = (1, b)$  and  $h$  and  $h'$  are undefined elsewhere. Clearly, both  $A$  and  $A'$  are partially specified. Then  $S = \{\varepsilon, a\}$  is a state cover of  $A$  and  $W = \{a\}$  is a characterisation set of  $A$ . Since  $A$  and  $A'$  have the same number of states (2), the *W-method* gives  $U_0 = S\Sigma[1]W = \{a, aa, ba, aaa, aba\}$ . However, the two DFSMs are  $U_0$ -equivalent but not equivalent. Indeed, the single input sequence which distinguishes between them is  $b$ , which is not included  $U_0$ .

This happens because, when  $A$  or  $A'$  are partially specified,  $A$  and  $A'$  may be  $\{s\}$ -equivalent for an input sequence  $s$ , but  $\{t\}$ -distinguishable for some prefix  $t$  of  $s$ . In Example 3.1,  $A$  and  $A'$  are  $\{ba\}$ -equivalent but  $\{b\}$ -distinguishable. Therefore, a solution is to take the set of all prefixes of  $U_k, \text{pref}(U_k)$ , instead of just  $U_k$ . In [7] it is shown that only a

subset  $U'_k$  of  $\text{pref}(U_k)$  is actually needed:

$$U'_k = U_k \cup S\Sigma[k]\Sigma = S\Sigma[k+1]W \cup S\Sigma[k]\Sigma.$$

In Example 3.1,  $A$  and  $A'$  are  $U'_0$ -distinguishable since  $b \in U'_0 = \{a, b, aa, ab, ba, aaa, aba\}$ . On the other hand, it will transpire that, when considering stream X-machines that are defined for every input sequence, the prefixes need not be included in the set of sequences derived from the partially specified associated automaton.

A variant of the  $W$ -method is the *partial W-method* ( $Wp$ -method) [20]. This reduces the size of the test suite at the expense of a slightly more complex generation algorithm. Instead of using the whole set  $W$  to check each state  $q$ , only a subset of this set can be used in certain cases. This subset  $W_q$  depends on the reached state  $q$  and is called an *identification set* of  $q$ . The  $Wp$ -method was originally devised for completely specified DFSMs [20], but can be extended to partially specified machines in a similar manner to the  $W$ -method [8].

#### 4. Stream X-machines

In essence, a stream X-machine is like a finite state machine but with one important difference: instead of abstract symbols, the transition labels are *processing functions*, which represent the elementary operations that the machine is capable of performing. Analogously to a finite state machine, a processing function will read inputs and produce outputs. Additionally, though, the machine has some internal store, called *memory*, so that the output produced by a processing function in response to an input will depend on the current memory value. Naturally, the processing function may also change the value of the memory. The model is formally described next.

**Definition 4.1.** A *stream X-Machine* (SXM) is a tuple  $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ , where

- $\Sigma$  is the finite *input alphabet*.
- $\Gamma$  is the finite *output alphabet*.
- $Q$  is the finite set of *states*.
- $M$  is a (possibly infinite) set called *memory*.
- $\Phi$  is a finite set of distinct *processing functions*; a processing function is a non-empty (partial) function of type  $M \times \Sigma \rightarrow \Gamma \times M$ .
- $F$  is the (partial) *next-state function*,  $F : Q \times \Phi \rightarrow Q$ .
- $q_0 \in Q$  is the initial state.
- $m_0 \in M$  is the initial memory value.

It is sometimes helpful to think of an X-machine as a finite automaton with the arcs labelled by functions from the set  $\Phi$ . The automaton  $A_Z = (\Phi, Q, F, q_0)$  over the alphabet  $\Phi$  is called *the associated finite automaton* (associated FA) of  $Z$ .  $A_Z$  is usually described by a state-transition diagram. Analogously to finite state machines, the function  $F$  may be extended to take sequences from  $\Phi^*$  to form the function  $F^*$ .

The set  $\Phi$  is often called the *type* of  $Z$ . Typically, each element of  $\Phi$  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  $\Phi$ .

**Example 4.1.** A stream X-machine  $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$  is used to describe the behaviour of a bounded stack having maximum  $k \geq 2$  elements of type  $E$ . The elements received by the machine are pushed on to the stack until this is full. The top most element of the stack is removed when the machine receives a special symbol, denoted by *rem*. The output symbols are considered to be the elements popped off the stack, a null symbol (used when a new element has been pushed on to the stack) and an error symbol, indicating that an operation (*push* or *pop*) has failed. Any failed operation will lead to an *Error* state; for simplicity, a special output symbol, *errid*, is also used to identify this state. The formal definition of  $Z$  is given in what follows:

- $\Sigma = E \cup \{\text{rem}\}$ , where  $E$  is a finite set and  $\text{rem} \notin E$ .
- $\Gamma = E \cup \{\text{null}, \text{error}, \text{errid}\}$ , where *null*, *error*, *errid* are pairwise distinct symbols such that  $\{\text{null}, \text{error}, \text{errid}\} \cap E = \emptyset$ .

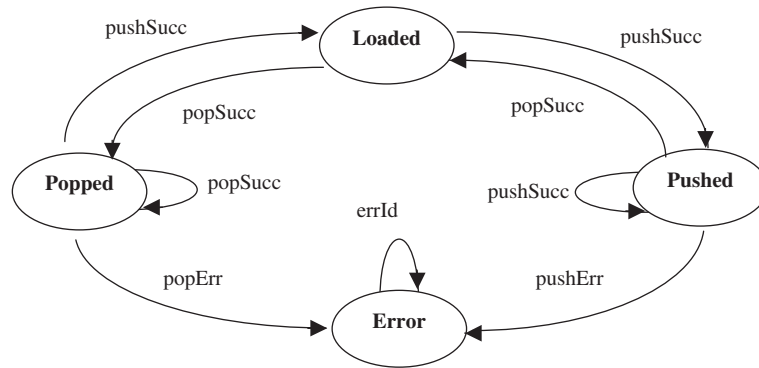


Fig. 1. The state-transition diagram of Z.

- $Q = \{Popped, Loaded, Pushed, Error\}$ . The initial state is *Popped*, i.e.  $q_0 = Popped$ .
- $M = E[k]$ . The memory holds the content of the stack. Initially, the stack is empty, i.e.  $m_0 = \varepsilon$ .
- $\Phi = \{pushSucc, popSucc, pushErr, popErr, errId\}$ , where *pushSucc* and *popSucc* denote the successful application of the *push* and *pop* operations, whilst *pushErr* and *popErr* denote their erroneous behaviour. *errId* is used to identify the *Error* state of the machine. The processing functions are defined in what follows. For simplicity, the definitions are restricted to the domains of the functions.

$$\begin{aligned}
 pushSucc(m, e) &= (m \ e, null), m \in M \setminus E^k, e \in E, \\
 popSucc(m \ e, rem) &= (m, e), m \in M \setminus E^k, e \in E, \\
 pushErr(m, e) &= (m, error), m \in E^k, e \in E, \\
 popErr(\varepsilon, rem) &= (\varepsilon, error), \\
 errId(m, \sigma) &= (m, errid), m \in M, \sigma \in \Sigma.
 \end{aligned}$$

- $F$  is as represented in Fig. 1. From the initial state *Popped*, two consecutive successful applications of *push* will always be possible. On the other hand, *pop* can either be successfully applied from *Popped* or produce an error, depending on the current value of the memory. Similarly, *pop* can be successfully applied twice from *Pushed*, whereas the application of *push* in the same state may either be successful or produce an error, depending on the value of the memory. Both *push* and *pop* are always successfully applied from the *Loaded* state.

The stream X-machine model is inspired by the Object machine described in [45] and will be used in illustrations later in the paper. It will transpire that the model is not suitable for test generation and, consequently, the X-machine will be redesigned in Section 12, but for the time being it serves our purpose.

A sequence  $p$  of processing functions induces a function  $\|p\|$  that shows the correspondence between a (memory, input sequence) pair and the (output sequence, memory) pair produced by the application, in turn, of the processing functions in the sequence  $p$ .

**Definition 4.2.** Given a sequence  $p \in \Phi^*$ ,  $p$  induces the (partial) function

$$\|p\| : M \times \Sigma^* \longrightarrow \Gamma^* \times M$$

defined as follows:

- $\|\varepsilon\|(m, \varepsilon) = (\varepsilon, m), m \in M,$
- Given  $p \in \Phi^*$  and  $\phi \in \Phi, \|p\phi\|(m, s\sigma) = (g\gamma, m')$ , for  $m, m' \in M, s \in \Sigma^*, g \in \Gamma^*, \sigma \in \Sigma, \gamma \in \Gamma$  such that there exists  $m'' \in M$  with  $\|p\|(m, s) = (g, m'')$  and  $\phi(m'', \sigma) = (\gamma, m')$ .

A machine computation takes the form of a traversal of all sequences of arcs in the state space from the initial state and the application, in turn, of the arc labels (which represent processing functions) 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 4.3.** Given an SXM  $Z$ , the *relation computed* by  $Z$ ,  $f_Z : \Sigma^* \longleftrightarrow \Gamma^*$ , is defined by:  $(s, g) \in f_Z$  if there exist  $p \in \Phi^*$  and  $m \in M$  such that  $(q_0, p) \in \text{dom } F^*$  and  $\|p\|(m_0, s) = (g, m)$ . We say that  $Z$  *computes*  $f_Z$ .

A completely defined SXM is one in which every sequence of inputs is processed by at least one sequence of functions accepted by the associated automaton.

**Definition 4.4.**  $Z$  is said to be completely defined if  $\text{dom } f_Z = \Sigma^*$ .

Sometimes, a stronger condition is required from a stream X-machine: a *completely specified* SXM is one in which there is at least one possible transition for every triplet  $q \in Q, m \in M, \sigma \in \Sigma$ .

**Definition 4.5.** An SXM  $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$  is called *completely specified* if for every  $q \in Q$ , every  $m \in M$  and every  $\sigma \in \Sigma$ , there exists  $\phi \in \Phi$  such that  $(m, \sigma) \in \text{dom } \phi$  and  $(q, \phi) \in \text{dom } F$ .

As not all memory values may actually be attained in every state of the machine (as explained in Section 5), it is sufficient for an SXM specification to be completely defined. On the other hand, it is often more convenient to produce SXM specifications that are completely specified, since, in this case, it is not necessary to determine whether a memory value can be attained in a given state. The SXM given in Example 4.1 is not completely specified since there is no erroneous *push* transition (when the stack is full) defined from *Popped*. Similarly, the erroneous *pop* transition (when the stack is empty) is not defined from *Pushed* and neither erroneous transitions exist from *Loaded*. On the other hand, it can be observed that the stack may never be full in *Popped* nor empty in *Pushed* and neither full nor empty in *Loaded*, so the machine is completely defined. A SXM may be transformed into one that is completely defined or completely specified by adding erroneous transitions to an *Error* state, in a similar way to Example 4.1.

A deterministic SXM is one in which there is at most one possible transition for any triplet  $q \in Q, m \in M, \sigma \in \Sigma$ .

**Definition 4.6.** A deterministic SXM (*DSXM*) is an SXM for which every two distinct processing functions that label arcs emerging from the same state have disjoint domains, i.e. for every  $\phi_1, \phi_2 \in \Phi$ , if there exists  $q \in Q$  such that  $(q, \phi_1), (q, \phi_2) \in \text{dom } F$  then either  $\phi_1 = \phi_2$  or  $\text{dom } \phi_1 \cap \text{dom } \phi_2 = \emptyset$ .

A deterministic SXM  $Z$  will compute a (partial) function  $f_Z$  (the function computed by  $Z$ ), rather than a relation. Note that the deterministic character of a SXM can be preserved by its transformation into a completely defined (or completely specified) SXM using erroneous transitions to an *Error* state. The SXM given in Example 4.1 is deterministic.

In the remainder of this paper we will only refer to deterministic SXMs.

## 5. Design for test conditions

When a specification is used as basis for test generation, it is natural to identify some design requirements that the specification will have to meet in order to facilitate the testing process. These are usually referred to as *design for test conditions*. Obviously, the weaker these conditions are, the more general the validity of the testing strategy will be. In the case of a DSXM specification, the design for test conditions place restrictions on the type  $\Phi$  of the specification. In this paper two design for test conditions will be required: output-distinguishability and input-uniformity.

Informally,  $\Phi$  is *output-distinguishable* when the output produced in response to any given input determines which processing function has been applied. That is, given two distinct processing functions  $\phi_1$  and  $\phi_2$ , a memory value  $m$  and an input  $\sigma$ , the two functions cannot produce the same output if given  $\sigma$  when the memory value is  $m$ . This property allows the tester to determine the sequence of processing functions applied by examining the output sequence produced when given an input sequence.

**Definition 5.1.**  $\Phi$  is called *output-distinguishable* if for all  $\phi_1, \phi_2 \in \Phi$ , whenever there exist  $m, m_1, m_2 \in M, \sigma \in \Sigma, \gamma \in \Gamma$  such that  $\phi_1(m, \sigma) = (\gamma, m_1)$  and  $\phi_2(m, \sigma) = (\gamma, m_2)$ , then  $\phi_1 = \phi_2$ .

The output-distinguishability condition can be enforced on a DSXM specification by making some memory variables observable (typically through debug messages in practical applications) and splitting some processing functions into two or more parts, in order to remove the overlapping of identical behaviour. In Example 4.1,  $\Phi$  is output-distinguishable.

Informally,  $\Phi$  is *input-uniform* if all memory values that are produced by the application of any single sequence of processing functions to any single memory are processed in a uniform way by any processing function—that is, any function can either process all such memory values or none. The memory values that are produced as the result of applying the same sequence of processing functions to the same starting memory value will be called *image-similar*. The memory values that are processed uniformly by any processing function will be called *domain-similar*.

**Definition 5.2.** Two memory values  $m_1, m_2 \in M$  are said to be *domain-similar* if for all  $\phi \in \Phi$ , there exists  $\sigma \in \Sigma$  such that  $(m_1, \sigma) \in \text{dom } \phi$  if and only if there exists  $\sigma \in \Sigma$  such that  $(m_2, \sigma) \in \text{dom } \phi$ .

Domain-similarity is an equivalence relation on  $M$ . For  $Z$  as defined in Example 4.1, domain-similarity induces three equivalence classes on  $M = E[k]$ :  $\{\varepsilon\}$ ,  $E[k - 1] \setminus \{\varepsilon\}$  and  $E^k$ .

**Definition 5.3.**  $isim_j$ ,  $j \geq 0$ , are relations on  $M$  defined as follows:

- $(m, m) \in isim_0$ ,  $m \in M$ .
- If  $j > 0$ ,  $(m_1, m_2) \in isim_j$ , for  $m_1, m_2 \in M$  such that
  - $(m_1, m_2) \in isim_{j-1}$  or
  - there exist  $m'_1, m'_2 \in M$  such that  $((m'_1, m'_2) \in isim_{j-1}$  and there exist  $\phi \in \Phi$ ,  $\sigma_1, \sigma_2 \in \Sigma$ ,  $\gamma_1, \gamma_2 \in \Gamma$ , such that  $(\phi(m'_1, \sigma_1) = (\gamma_1, m_1)$  and  $\phi(m'_2, \sigma_2) = (\gamma_2, m_2))$ ).

Two memory values  $m_1, m_2 \in M$  are said to be *image-similar* if  $(m_1, m_2) \in isim_j$  for some  $j \geq 0$ .

A direct consequence of the above definition is that if there exists  $j_0 \geq 0$  such that  $isim_{j_0} = isim_{j_0+1}$  then  $isim_{j_0} = isim_{j_0+i}$ ,  $i \geq 1$ , so the image-similarity relation coincides with  $isim_{j_0}$ .

For  $Z$  as defined in Example 4.1,  $0 \leq j \leq k$  and  $m_1, m_2 \in E[k]$ ,  $(m_1 m_2) \in isim_j$  if and only if  $|m_1| = |m_2|$  and, if  $|m_1| > j$ , the bottom-most  $|m_1| - j$  elements of  $m_1$  and  $m_2$  coincide. Thus,  $(m_1 m_2) \in isim_k$  if and only if  $|m_1| = |m_2|$ . Furthermore, it can be observed that  $isim_k = isim_{k+1}$ . Therefore  $m_1$  and  $m_2$  are image-similar if and only if  $|m_1| = |m_2|$ .

An equivalent form of Definition 5.3 is given by the following lemma.

**Lemma 5.1.** Two memory values  $m_1, m_2 \in M$  are image-similar if and only if there exist  $p \in \Phi^*$ ,  $m \in M$ ,  $s_1, s_2 \in \Sigma^*$ ,  $g_1, g_2 \in \Gamma^*$ , such that  $(\|p\|(m, s_1) = (g_1, m_1)$  and  $\|p\|(m, s_2) = (g_2, m_2))$ .

**Proof.** By induction on  $j \geq 0$  it follows that  $(m_1, m_2) \in isim_j$  if and only if there exist  $p \in \Phi[j]$ ,  $m \in M$ ,  $s_1, s_2 \in \Sigma^*$ ,  $g_1, g_2 \in \Gamma^*$ , such that  $(\|p\|(m, s_1) = (g_1, m_1)$  and  $\|p\|(m, s_2) = (g_2, m_2))$ .  $\square$

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-similarity (Definition 5.4).

**Definition 5.4.**  $\Phi$  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  $\Phi$  is input-uniform, one can determine an input sequence that drives a sequence of processing functions by simply selecting appropriate input symbols for each processing function in the sequence, one at a time, without needing to know the processing functions to be applied next. This is conveyed by the following lemma.

**Lemma 5.2.** Suppose  $\Phi$  is input-uniform. Let  $\phi_1, \dots, \phi_j \in \Phi$ ,  $j \geq 2$ , for which there exists  $s = \sigma_1 \dots \sigma_j \in \Sigma^*$  such that  $(m_0, s) \in \text{dom } \|\phi_1 \dots \phi_j\|$ . Then, for every  $i$ ,  $1 \leq i \leq j - 1$ , and every  $\sigma'_1, \dots, \sigma'_i \in \Sigma$  such that  $(m_0, \sigma'_1 \dots \sigma'_i) \in \text{dom } \|\phi_1 \dots \phi_i\|$  there exist  $\sigma'_{i+1}, \dots, \sigma'_j \in \Sigma$  such that  $(m_0, \sigma'_1 \dots \sigma'_j) \in \text{dom } \|\phi_1 \dots \phi_j\|$ .

**Proof.** Let  $\|\phi_1 \dots \phi_k\|(m_0, \sigma_1 \dots \sigma_k) = (\gamma_1 \dots \gamma_k, m_k)$  and  $\|\phi_1 \dots \phi_k\|(m_0, \sigma'_1 \dots \sigma'_k) = (\gamma'_1 \dots \gamma'_k, m'_k)$ ,  $m_k, m'_k \in M$ ,  $\gamma_k, \gamma'_k \in \Gamma$ ,  $1 \leq k \leq i$ . By Lemma 5.1,  $m_k$  and  $m'_k$  are image-similar. Thus there exists  $\sigma'_{i+1} \in \Sigma$  such that  $(m_0, \sigma'_1, \dots, \sigma'_{i+1}) \in \text{dom } \|\phi_1 \dots \phi_{i+1}\|$ . Since the existence of  $\sigma'_{i+1}$  can be deduced from the existence of  $\sigma'_1, \dots, \sigma'_i$ , the required result follows by induction on  $j \geq i + 1$ .  $\square$

In the worst case, input-uniformity can be achieved by designing processing functions that are triggered by single inputs—in this case every memory value is only similar to itself.

A stronger variant of this condition, *input-completeness*, requires every processing function to be able to process all memory values. When  $\Phi$  is input-complete, all memory values are pairwise domain-similar.

**Definition 5.5.**  $\Phi$  is called *input-complete* if for all  $\phi \in \Phi$ , all  $m \in M$ , there exists  $\sigma \in \Sigma$  such that  $(m, \sigma) \in \text{dom } \phi$ .

In Example 4.1,  $\Phi$  is input-uniform but not input-complete (*popSucc* does not apply to the empty stack, for example). The stronger input-completeness condition appears to be required in all previous publications addressing stream X-machine based testing [2,5,11,25,26,28,29,31,33–38].

Analogously to DFSMs, testing from a DSXM specification  $Z$  will involve the selection of input sequences that *reach* and *distinguish* the states of  $Z$ . These two issues are discussed in the following section.

### 6. Reaching and distinguishing states in a DSXM

As the labels used in the state-transition diagram of a DSXM are functions rather than mere symbols, there may be states that are reachable in the diagram but cannot actually 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 functions that distinguish between them can never be applied.

**Example 6.1.** Consider  $Z_1 = (\{a, b\}, \{a, b\}, \{0, 1, 2, 3\}, \{0, 1\}, \{f, g\}, F_1, \{0\}, \{0\})$ , with  $f(0, a) = (a, 1)$  and  $g(0, b) = (b, 1)$  ( $f$  and  $g$  are undefined elsewhere) and  $F_1$  as represented in Fig. 2. As the domains of both  $\|ff\|$  and  $\|gg\|$  are empty, state 3 cannot be reached by any input sequence applied to  $Z$ . Furthermore, since neither  $f$  can be applied in state 1, nor  $g$  can be applied in state 2, states 1 and 2 cannot be distinguished by any input even though they are distinguishable states in the associated FA.

#### 6.1. Realisable sequences

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

**Definition 6.1.** Given a memory value  $m \in M$ , the set  $R_\Phi(m)$  is defined to consist of all sequences of processing functions  $p = \phi_1 \dots \phi_n \in \Phi^*$ ,  $n \geq 0$ , for which there exists  $s = \sigma_1 \dots \sigma_n \in \Sigma^*$  such that  $(m, s) \in \text{dom } \|p\|$ .

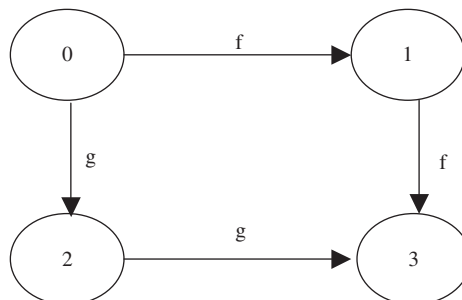


Fig. 2. The state-transition diagram of  $Z_1$ .

**Definition 6.2.** Given a state  $q \in Q$  and a memory value  $m \in M$ , a sequence of processing functions  $p \in \Phi^*$  is said to be *realisable* in  $q$  and  $m$  if  $p \in L_{AZ}(q)$  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 processing functions realisable (in  $q$  and  $m$ ) is denoted by  $LR_Z$  (or  $LR_Z(q, m)$ ), i.e.  $LR_Z(q, m) = L_{AZ}(q) \cap R_\Phi(m)$  and  $LR_Z = L_{AZ} \cap R_\Phi(m_0)$ .

Note that, when  $\Phi$  is input-complete, all sequences of processing functions accepted by the associated automaton are realisable. Such DSXMs are called *controllable*.

**Definition 6.3.**  $Z$  is said to be *controllable* if  $LR_Z = L_{AZ}$ .

### 6.2. $r$ -reachable states

Sequences in  $LR_Z$  make it possible to reach some states of a DSXM using appropriate input sequences. Such states will be referred to as  *$r$ -reachable*.

**Definition 6.4.** A state  $q \in Q$  is said to be  *$r$ -reachable* if there exists  $p \in LR_Z$  such that  $F^*(q_0, p) = q$ .

Any state that is not  *$r$ -reachable* can be removed without affecting the function computed by the machine. Since  $\varepsilon \in LR_Z$ , the initial state is always  *$r$ -reachable*.

An  *$r$ -state cover* is a minimal set of realisable sequences  $S_r$ ,  $\varepsilon \in S_r$ , that reaches every  *$r$ -reachable* state in  $Z$ .

**Definition 6.5.** A set  $S_r \subseteq LR_Z$  is called an  *$r$ -state cover* of  $Z$  if:

- $\varepsilon \in S_r$ .
- For every  *$r$ -reachable* state  $q$  of  $Z$  there exists  $p \in S_r$  such that  $F^*(q_0, p) = q$ .
- For every two distinct sequences  $p_1, p_2 \in S_r$ ,  $F^*(q_0, p_1) \neq F^*(q_0, p_2)$ .

For  $Z$  as in Example 4.1,  $\varepsilon, popErr, pushSucc, pushSucc pushSucc \in LR_Z$ . Thus all states of  $Z$  are  *$r$ -reachable* and  $S_r = \{\varepsilon, popErr, pushSucc, pushSucc pushSucc\}$  is an  *$r$ -state cover* of  $Z$ .

### 6.3. Attainable memory values

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

**Definition 6.6.** Given a state  $q \in Q$ , a memory value  $m \in M$  is said to be *attainable* in  $q$  if there exist  $p \in LR, s \in \Sigma^*, g \in \Gamma^*$  such that  $F^*(q_0, p) = q$  and  $p(m_0, s) = (g, m)$ . The set of all memory values attainable in  $q$  is denoted by  $MAtt(q)$ .

For  $Z$  as in Example 4.1,  $MAtt(Popped) = E[k-2], MAtt(Loaded) = E[k-1] \setminus \{\varepsilon\}, MAtt(Pushed) = E[k] \setminus E[1], MAtt(Error) = \{\varepsilon\} \cup E^k$ .

### 6.4. $r$ -distinguishable states

We will say that two states  $q_1$  and  $q_2$  of a DSXM are  *$r$ -distinguishable* if it is possible to distinguish between them by applying a finite set of realisable sequences of processing functions in any attainable memory value of  $q_1$  and  $q_2$ , respectively.

**Definition 6.7.** Given  $q_1, q_2 \in Q$ , a set  $Y \subseteq \Phi^*$  is said to  *$r$ -distinguish* between  $q_1$  and  $q_2$  if for every  $m_1 \in MAtt(q_1)$  and every  $m_2 \in MAtt(q_2)$ ,  $LR(q_1, m_1) \cap Y \neq LR(q_2, m_2) \cap Y$ . Two states  $q_1$  and  $q_2$  are said to be  *$r$ -distinguishable* if there exists a *finite* set of sequences  $Y$  that  *$r$ -distinguishes* between them.

As shown by Example 6.1, not every pair of states of a DSXM can necessarily be  *$r$ -distinguished* by a set of sequences even if the associated FA is minimal. Furthermore, even if such a set exists, it may not be finite, as shown by the following counterexample.

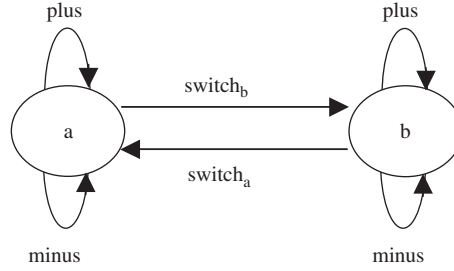


Fig. 3. The state-transition diagram of  $Z_2$ .

**Example 6.2.** Consider  $Z_2 = (\{-1, 1\}, \{m, p, a, b\}, \{a, b\}, N_0, \Phi_2, F_2, \{a\}, \{0\})$ , where  $N_0$  is the set of non-negative integers;  $\Phi_2 = \{minus, plus, switch_a, switch_b\}$ ;  $minus(n, -1) = (m, n - 1), n \geq 1$ ;  $plus(n, 1) = (p, n + 1), n \geq 0$ ;  $switch_a(0, -1) = (a, 0)$ ;  $switch_b(0, -1) = (b, 0)$  (the processing functions are undefined elsewhere) and  $F_2$  is as represented in Fig. 3. Then  $MAtt(a) = MAtt(b) = N_0$  and for every  $n_1, n_2 \in N_0, minus^{n_1} switch_b \in LR(a, n_1) \setminus LR(b, n_2)$ . However, for every  $n \in N_0$  and every  $n_1, n_2 \in N_0$  such that  $n_1, n_2 > n, LR(a, n_1) \cap \Phi_2[n] = LR(b, n_2) \cap \Phi_2[n] = \{minus, plus\}[n]$ . Thus  $a$  and  $b$  are not  $r$ -distinguishable.

For  $Z$  as in Example 4.1,  $\{errId\}$   $r$ -distinguishes between *Error* and any of *Popped*, *Loaded* and *Pushed*, so *Error* is  $r$ -distinguishable from any other state of  $Z$ . The problem of distinguishing between the remaining states is slightly more complex. First, it can be observed that for every  $q \in \{Popped, Loaded, Pushed\}$  and every  $m \in MAtt(q), LR_Z(q, m) = K_1 \cup K_2 \cup K_3$ , where

- $K_1$  consists of all sequences  $p \in \{pushSucc, popSucc\}^*$  for which every prefix  $p'$  of  $p$  satisfies  $0 \leq N_{pushSucc}(p') - N_{popSucc}(p') + |m| \leq k$ , where  $N_\phi(p')$  denotes the number of occurrences of  $\phi$  in the sequence  $p'$ ;
- $K_2$  consists of all sequences  $p = p_1, p_2$ , with  $p_1 \in K_1$  such that  $N_{pushSucc}(p_1) - N_{popSucc}(p_1) + |m| = 0$  and  $p_2 \in \{popErr\}\{errId\}^*$ ;
- $K_3$  consists of all sequences  $p = p_1, p_2$ , with  $p_1 \in K_1$  such that  $N_{pushSucc}(p_1) - N_{popSucc}(p_1) + |m| = k$  and  $p_2 \in \{pushErr\}\{errId\}^*$ .

Hence, it follows that for every  $q_1, q_2 \in \{Popped, Loaded, Pushed\}$ , every  $m_1 \in MAtt(q_1)$  and every  $m_2 \in MAtt(q_2), LR_Z(q_1, m_1) = LR_Z(q_2, m_2)$  if and only if  $|m_1| = |m_2|$ . As  $M$  is finite,  $q_1$  and  $q_2$  are  $r$ -distinguishable if and only if for every  $m_1 \in MAtt(q_1)$  and every  $m_2 \in MAtt(q_2), LR_Z(q_1, m_1) \neq LR_Z(q_2, m_2)$ . Consequently,  $q_1$  and  $q_2$  are  $r$ -distinguishable if and only if for every  $m_1 \in MAtt(q_1)$  and every  $m_2 \in MAtt(q_2), |m_1| \neq |m_2|$ . Since  $MAtt(Popped) = E[k - 2], MAtt(Loaded) = E[k - 1] \setminus \{\varepsilon\}$  and  $MAtt(Pushed) = E[k] \setminus E[1]$ , the following three cases can be distinguished:

- $k = 2$ : In this case  $MAtt(Popped) = \{\varepsilon\}, MAtt(Loaded) = E$  and  $MAtt(Pushed) = E^2$ . Thus, *Popped*, *Loaded* and *Pushed* are pairwise  $r$ -distinguishable. Since  $popErr \in LR_Z(Popped, \varepsilon)$  and  $popErr$  is allowed neither from *Loaded* nor from *Pushed*,  $\{popErr\}$   $r$ -distinguishes *Popped* from any of *Pushed* and *Loaded*. Similarly,  $\{pushErr\}$   $r$ -distinguishes *Pushed* from any of *Popped* and *Loaded*.
- $k = 3$ : In this case  $MAtt(Popped) = E \cup \{\varepsilon\}, MAtt(Loaded) = E^2 \cup E$  and  $MAtt(Pushed) = E^3 \cup E^2$ . Thus, *Popped* and *Pushed* are  $r$ -distinguishable, but *Loaded* can be  $r$ -distinguished neither from *Popped*, nor from *Pushed*. Since  $popErr \in LR_Z(Popped, \varepsilon)$  and for every  $m \in E, popSucc popErr \in LR(Popped, m)$  and neither of these two sequences can be applied from *Pushed*,  $\{popErr, popSucc popErr\}$   $r$ -distinguishes between *Popped* and *Pushed*.
- $k \geq 4$ : In this case  $MAtt(Popped) = E^{k-2} \cup \dots \cup E^2 \cup E \cup \{\varepsilon\}, MAtt(Loaded) = E^{k-1} \cup \dots \cup E^3 \cup E^2 \cup E$  and  $MAtt(Pushed) = E^k \cup \dots \cup E^4 \cup E^3 \cup E^2$ . Thus,  $\{Popped, Loaded, Pushed\}$  contains no pairwise  $r$ -distinguishable elements.

An  $r$ -characterisation set is a set of sequences of processing functions that  $r$ -distinguishes between every pair of  $r$ -distinguishable states.

**Definition 6.8.** A set  $W_r \subseteq \Phi^*$  is called an  $r$ -characterisation set of  $Z$  if  $W_r$   $r$ -distinguishes between every two  $r$ -distinguishable states of  $Z$ .

For  $Z$  as defined in Example 4.1, three cases will have to be considered, as discussed earlier:

- $k = 2$  :  $W_r = \{errId, pushErr, popErr\}$  is an  $r$ -characterisation set of  $Z$ .
- $k = 3$  :  $W_r = \{errId, popErr, popSucc popErr\}$  is an  $r$ -characterisation set of  $Z$ .
- $k \geq 4$  :  $W_r = \{errId\}$  is an  $r$ -characterisation set of  $Z$ .

## 7. Testing against a DSXM specification: prerequisites

The remainder of the paper will address the problem of deriving test suites from a DSXM specification. A test suite is a finite set of input sequences that can establish the *correctness* of the implementation under test with respect to the specification. Naturally, functional equivalence is used as the notion of correctness. This section states the requirements that the specification will have to meet and the assumptions made about the implementation under test.

### 7.1. The specification

The specification considered will be a completely defined DSXM  $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ , with  $\Phi$  output-distinguishable and input-uniform (the design for test conditions). Unlike in previous publications addressing stream X-machine based testing,  $Z$ , may be non-controllable. As the specification is required to be completely defined, when inputs are not expected in a state for certain attainable memory values, appropriate erroneous transitions to an error state will be introduced.

### 7.2. The fault domain

When testing against a formal specification, the IUT is normally considered to be functionally equivalent to some element from a set of models, called the fault domain, which is determined by the assumptions one can make about the implementation. As the specification is a DSXM, naturally, the fault domain will contain DSXMs, so it will be assumed that the IUT behaves like some unknown completely defined DSXM  $Z'$  with the same input alphabet and output alphabet as the specification  $Z$ . Since the memory models the data and the internal variables used by the implementation,  $Z'$  will have the same memory as  $Z$ . Naturally,  $Z$  and  $Z'$  will be initialised with the same values for the memory. Note, however, that the testing procedure will not depend on the choice of the initial memory. A different initial memory, or initial state, may produce different test data, but the test generation algorithm will remain unchanged.

Additionally, when testing from a DSXM, it is normally assumed that  $Z$  and  $Z'$  have the same sets of processing functions (type) [2,5,11,25,26,28,29,33–38]. This corresponds to the situation where the implementation is built either from reusable trusted components or from components that have been thoroughly tested.

As for finite automata, the number of the states of the implementation model will be bounded by an integer  $n'$  estimated by the tester, which is greater than or equal to the number of states  $n$  of the specification. However, an upper bound on the number of states of  $Z'$  is not a sufficient criterion for limiting the length of the test sequence unless  $Z'$  is controllable (otherwise, for a sufficiently large memory, there may still be states that cannot be reached and transitions that cannot be checked by any input sequence). When the memory is finite, as is always the case in practical applications, a controllable stream X-machine model of the implementation will always exist; in the worst case, this can be found by constructing an equivalent machine whose states are the (state, memory) pairs of the original model. Obviously, such a construction will lead to an extremely large upper bound  $n'$ . In such cases, additional bounds may be used to limit the size of the test suite, as explained in the penultimate section of the paper. For the time being, however, the test selection strategy will only be based on the upper bound  $n'$  of a controllable model of the implementation.

Consequently, it will be assumed that the IUT can be modelled by a completely defined, controllable, DSXM  $Z' = (\Sigma, \Gamma, Q', M, \Phi, F', q'_0, m_0)$  having at most  $n'$  states. The fault domain will contain all such DSXMs; among them, at least one will be functionally equivalent to the specification. The question, how we can characterise when there exists at least one controllable DSXM  $Z'$  for which  $f_Z = f_{Z'}$ , is obviously worth investigating.

**Example 7.1.** Consider  $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$  as defined in Example 4.1 and  $Z' = (\Sigma, \Gamma, Q', M, \Phi, F', q'_0, m_0)$  with  $Q = \{0, \dots, k\} \cup \{Error\}$ ,  $q'_0 = 0$  and  $F'$  defined by

$$\begin{aligned} F'(0, pushSucc) &= 1, F'(0, popErr) = Error; \\ F'(i, pushSucc) &= i + 1, F'(i, popSucc) = i - 1, 1 \leq i \leq k - 1; \\ F'(k, pushErr) &= Error, F'(0, popSucc) = k - 1; \\ F'(Error, errId) &= Error \text{ (} F' \text{ is undefined elsewhere).} \end{aligned}$$

It can be observed that  $Z'$  is controllable and  $f_Z = f_{Z'}$ . Thus the test generation problem can be formulated for  $n' \geq k + 2$ .

Under the above stated conditions, a set of test sequences will be generated that, for every  $Z'$  in the fault domain, can establish whether  $Z'$  is functionally equivalent to  $Z$ .

The concepts and results necessary for the construction of the test suite will be gradually introduced in the following four sections. First, Section 8 shows how checking the functional equivalence of the two machines can be reduced to exploring the relationship between their associated FAs. This, in turn, will be investigated through the use of a product machine, defined in Section 9. Next, Section 10 will introduce the concept of a test function as a means of converting sequences of processing functions derived from the product machine into input sequences. Finally, the generation of the test suite will be described in Section 11.

## 8. Checking realisable sequences

As  $Z$  and  $Z'$  are assumed to have the same type and this is output-distinguishable, the testing process will have to check that the sequences of processing functions allowed by the implementation correspond to those specified. Only the realisable sequences will have to be considered, as the others have no functional role. This idea is captured by the following lemma.

**Lemma 8.1.**  $f_Z = f_{Z'}$  if and only if  $LR_Z = LR_{Z'}$ .

**Proof.** “ $\Leftarrow$ ”: Follows from Definition 4.3.

“ $\Rightarrow$ ”: Let  $p = \phi_1 \dots \phi_k \in LR_Z$ . 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  $\phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i)$ ,  $1 \leq i \leq k$ . Since  $f_Z(\sigma_1 \dots \sigma_k) = f_{Z'}(\sigma_1 \dots \sigma_k)$ , 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  $\phi'_i(m'_{i-1}, \sigma_i) = (\gamma_i, m'_i)$ ,  $1 \leq i \leq k$ , where  $m'_0 = m_0$ . Since  $\Phi$  is output-distinguishable, 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,  $LR_Z \subseteq LR_{Z'}$ . Similarly,  $LR_{Z'} \subseteq LR_Z$ , so  $LR_Z = LR_{Z'}$ .  $\square$

Furthermore, since  $Z$  and  $Z'$  are completely defined, it is sufficient just to check that every realisable sequence of processing functions in  $Z'$  can also be found in  $Z$  (or vice versa).

**Lemma 8.2.**  $LR_Z = LR_{Z'}$  if and only if  $LR_{Z'} \subseteq LR_Z$ .

**Proof.** We provide a proof by contradiction. Assume  $LR_{Z'} \subset LR_Z$ . Let  $p = \phi_1 \dots \phi_k \in LR_Z \setminus LR_{Z'}$ . Then there exists  $i$ ,  $1 \leq i \leq k - 1$ , such that  $\phi_1 \dots \phi_i \in LR_{Z'}$  and  $\phi_1 \dots \phi_i \phi_{i+1} \notin LR_{Z'}$ . Since  $\phi_1 \dots \phi_i \phi_{i+1} \in R_\Phi(m_0)$ , there exist  $\sigma_1, \dots, \sigma_{i+1} \in \Sigma$ ,  $\gamma_1, \dots, \gamma_{i+1} \in \Gamma$ ,  $m_1 \dots m_{i+1} \in M$  such that  $\|\phi_1 \dots \phi_i\|(m_0, \sigma_1 \dots \sigma_j) = (\gamma_1 \dots \gamma_j, m_j)$ ,  $1 \leq j \leq i + 1$ . Since  $Z'$  is completely defined and  $\phi_1 \dots \phi_i \phi_{i+1} \notin LR_{Z'}$ , there exists  $\phi'_{i+1} \in \Phi$  with  $(m_i, \sigma_{i+1}) \in \text{dom } \phi'_{i+1}$  such that  $\phi_1 \dots \phi_i \phi'_{i+1} \in LR_{Z'}$ . Since  $\text{dom } \phi_{i+1} \cap \text{dom } \phi'_{i+1} \neq \emptyset$  and  $Z$  is deterministic,  $\phi_1 \dots \phi_i \phi'_{i+1} \notin LR_Z$ . Thus  $\phi_1 \dots \phi_i \phi'_{i+1} \in LR_{Z'} \setminus LR_Z$ , so  $LR_{Z'} \subset LR_Z$  does not hold, which is a contradiction.  $\square$

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

**Lemma 8.3.**  $LR_Z = LR_{Z'}$  if and only if  $L_{A_{Z'}} \subseteq L_{A_Z}$ .

**Proof.** We prove that  $LR_{Z'} \subseteq LR_Z$  if and only if  $L_{A_{Z'}} \subseteq L_{A_Z}$ . Assume  $L_{A_{Z'}} \subseteq L_{A_Z}$ . Then  $L_{A_{Z'}} \cap R_\Phi(m_0) \subseteq L_{A_Z} \cap R_\Phi(m_0)$ . Thus  $LR_{Z'} \subseteq LR_Z$ . Conversely, assume  $LR_{Z'} \subseteq LR_Z$ . Then  $L_{A_{Z'}} \cap R_\Phi(m_0) \subseteq L_{A_Z} \cap R_\Phi(m_0)$ . Since  $Z'$  is controllable,  $L_{A_{Z'}} \cap R_\Phi(m_0) = L_{A_{Z'}}$ , so  $L_{A_{Z'}} \subseteq L_{A_Z} \cap R_\Phi(m_0)$ . Thus  $L_{A_{Z'}} \subseteq L_{A_Z}$ .  
 By Lemma 8.2,  $LR_Z = LR_{Z'}$  if and only if  $LR_{Z'} \subseteq LR_Z$ . Thus, the result follows.  $\square$

## 9. The product machine

As  $L_{A_Z}$  may contain sequences that are not realisable, the  $W$ -method cannot be directly applied to establish the equivalence of the two associated finite automata. Instead, a state-counting approach will be used, which involves the construction of the product machine of  $Z$  and  $Z'$ . Given two FAs,  $A_Z$  and  $A_{Z'}$ , one can 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 FAs. 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  $F'(q', \phi) = q'_1$  exist in  $A_Z$  and  $A_{Z'}$ , respectively. The result of such a construction corresponds to the intersection of the languages accepted by the two FAs. If the languages accepted by  $A_Z$  and  $A_{Z'}$  are different, then there will be a transition from some  $(q, q')$  which only one of the two FAs can follow. By adding to the cross-product FA an extra state, *Fail*, and transitions  $F_P((q, q'), \phi) = \text{Fail}$  to correspond to transitions which can be taken by  $Z'$  but not by  $Z$ , testing for inclusion of  $L_{A_{Z'}}$  into  $L_{A_Z}$  will correspond to testing of the cross-product FA in order to find if the *Fail* state is reachable. If the two DSXMs,  $Z$  and  $Z'$ , are considered instead of their FAs, this construction will give rise to the *product machine*.

**Definition 9.1.** The *product machine* formed from  $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$  and  $Z' = (\Sigma, \Gamma, Q', M, \Phi, F', q'_0, m_0)$  is the DSXM  $P(Z, Z') = (\Sigma, \Gamma, Q_P, M, \Phi, F_P, (q_0, q'_0), m_0)$  in which  $Q_P = (Q \times Q') \cup \{\text{Fail}\}$ ,  $\text{Fail} \notin Q \times Q'$ , and  $F_P$  is defined by the following rules:

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

It can be verified that  $P(Z, Z')$  is a deterministic SXM. Furthermore, since  $Z'$  is controllable and every sequence of processing functions accepted by  $A_{P(Z, Z')}$  is also accepted by  $A_{Z'}$ ,  $P(Z, Z')$  is also controllable. Note also that, unlike  $Z$  and  $Z'$ ,  $P(Z, Z')$  may not be completely defined, since no processing function can be applied from *Fail*. 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 testing for functional equivalence of  $Z$  and  $Z'$  corresponds to establishing that the *Fail* state of the product machine is not reachable.

**Lemma 9.1.** 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  $F'^*(q'_0, p) = q'$ .

**Proof.** Follows from Definition 9.1 by induction on the length of  $p$ .  $\square$

**Lemma 9.2.** Given  $p \in \Phi^*$ ,  $p$  reaches *Fail* in  $A_{P(Z, Z')}$  if and only if  $p \in L_{A_{Z'}} \setminus L_{A_Z}$  and  $p = p_1\phi$  for some  $p_1 \in L_{A_Z} \cap L_{A_{Z'}}$  and  $\phi \in \Phi$ .

**Proof.** By Definition 9.1,  $p$  reaches *Fail* in  $A_{P(Z, Z')}$  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  $F_P((q, q'), \phi) = \text{Fail}$ . By Lemma 9.1,  $F_P^*((q_0, q'_0), p_1) = (q, q')$  if and only if  $F^*(q_0, p_1) = q$ ,  $F'^*(q'_0, p_1) = q'$ . By Definition 9.1,  $F_P((q, q'), \phi) = \text{Fail}$  if and only if  $(q, \phi) \notin \text{dom } F$  and  $(q', \phi) \in \text{dom } F'$ . Thus,  $p$  reaches *Fail* in  $A_{P(Z, Z')}$  if and only if  $p \in L_{A_{Z'}} \setminus L_{A_Z}$  and  $p = p_1\phi$  for some  $p_1 \in L_{A_Z} \cap L_{A_{Z'}}$  and  $\phi \in \Phi$ .  $\square$

**Lemma 9.3.** *Fail* is not reachable in  $A_{P(Z, Z')}$  if and only if  $L_{A_{Z'}} \subseteq L_{A_Z}$ .

**Proof.**  $L_{A_{Z'}} \subseteq L_{A_Z}$  does not hold if and only if there exist  $p \in \Phi^*$ ,  $\phi \in \Phi$  such that  $p \in L_{A_Z} \cap L_{A_{Z'}}$  and  $p\phi \in L_{A_{Z'}} \setminus L_{A_Z}$ . Thus, by Lemma 9.2, *Fail* is reachable in  $A_{P(Z,Z')}$  if and only if  $L_{A_{Z'}} \subseteq L_{A_Z}$  does not hold.  $\square$

**Lemma 9.4.** *Fail* is not reachable in  $A_{P(Z,Z')}$  if and only if  $LR_Z = LR_{Z'}$ .

**Proof.** Follows from Lemmas 9.3 and 8.3.  $\square$

**Lemma 9.5.** *Fail* is not reachable in  $A_{P(Z,Z')}$  if and only if  $f_Z = f_{Z'}$ .

**Proof.** Follows from Lemmas 9.4 and 8.1.  $\square$

## 10. Test function

Suppose we have generated appropriate sequences of processing functions to check whether the *Fail* state of  $A_{P(Z,Z')}$  is reachable. We will then need a mechanism that translates sequences of processing functions into sequences of inputs. This will be called a *test function* of  $Z$ .

**Definition 10.1.** Suppose  $\Phi$  is input-uniform. A *test function* of  $Z$  is a function  $t : \Phi^* \rightarrow \Sigma^*$  that satisfies the following conditions:

- $t(\varepsilon) = \varepsilon$ . (1)
- Let  $p \in \Phi^*$  and  $\phi \in \Phi$ .
  - Suppose  $p \in L_{A_Z}$  and  $(m_0, t(p)) \in \text{dom } \|p\|$ . Let  $\|p\|(m_0, t(p)) = (g, m)$ ,  $g \in \Gamma^*$ ,  $m \in M$ .
    - \* If there exists  $\sigma \in \Sigma$  such that  $(m, \sigma) \in \text{dom } \phi$  then  $t(p\phi) = t(p)\sigma$ , for some  $\sigma$  that satisfies this condition. (2)
    - \* Else,  $t(p\phi) = t(p)$ . (3)
  - Otherwise,  $t(p\phi) = t(p)$ . (4)

The first rule (1) is the base case, stating that the empty path is transformed into the empty input sequence, while the remaining three rules are recursive cases, explaining how  $t(p\phi)$  may be defined in terms of  $t(p)$ . Rules (2) and (3) give the case where  $p \in LR_Z$ :  $t(p)$  is extended by some value  $\sigma$  that triggers  $\phi$  if such a value exists, otherwise  $t(p)$  is left unchanged. The final rule (4) states that  $t(p)$  need not be extended once it has been determined that  $p \notin LR_Z$ . In general, a test function of  $Z$  is not uniquely determined, many different test functions may exist.

For  $Z$  as defined in Example 4.1 and  $k = 3$ , consider the sequence  $\text{pushSucc}^5$ . First, the second rule will be applied three times, so

$$\begin{aligned} t(\text{pushSucc}) &= e_1, \\ t(\text{pushSucc}^2) &= e_1e_2, \\ t(\text{pushSucc}^3) &= e_1e_2e_3, \end{aligned}$$

for some  $e_1, e_2, e_3 \in E$ . Then the third rule will be applied, so

$$t(\text{pushSucc}^4) = e_1e_2e_3.$$

Finally, by the last rule,

$$t(\text{pushSucc}^5) = e_1e_2e_3.$$

On the other hand, for the sequence  $\text{pushSucc}^3 \text{errId}^2$ , the second rule will be applied four times, so

$$t(\text{pushSucc}^3 \text{errId}) = e_1e_2e_3\sigma,$$

for some  $\sigma \in \Sigma$ . Similarly, by the last rule,

$$t(\text{pushSucc}^3 \text{errId}^2) = e_1e_2e_3\sigma.$$

In general, consider a sequence of processing functions,  $p = \phi_1 \dots \phi_k$ . If  $p$  is contained in  $LR_Z$  then the input-uniformity condition will ensure that the second rule of the definition will be applied for every processing function in the sequence, so  $t(p)$  will be an input sequence  $\sigma_1 \dots \sigma_k$  that drives  $p$ . Otherwise, the second rule will be applied until a prefix  $p_1 = \phi_1 \dots \phi_j$  of  $p$  is found that is not in  $LR_Z$ . If  $p_1 \in R_\Phi(m_0)$ , then the second rule will be applied once more, so  $t(p_1) = \sigma_1 \dots \sigma_j$ , otherwise the third rule will be used and  $t(p_1) = \sigma_1 \dots \sigma_{j-1}$ . Then the last rule will be applied and the input sequence will extended no further, so  $t(p) = t(p_1)$ .

**Lemma 10.1.** *Suppose  $\Phi$  is input-uniform. Let  $t$  be a test function of  $Z$  and  $\phi_1, \dots, \phi_k \in \Phi$ .*

- *If  $\phi_1 \dots \phi_k \in LR_Z$  then  $t(\phi_1 \dots \phi_k) = \sigma_1 \dots \sigma_k$ , for some  $\sigma_1, \dots, \sigma_k \in \Sigma$  such that  $(m_0, \sigma_1 \dots \sigma_k) \in \text{dom } \|\phi_1 \dots \phi_k\|$ .*
- *If there is some  $j, 1 \leq j \leq k - 1$ , such that  $\phi_1 \dots \phi_j \in LR_Z$  and  $\phi_1 \dots \phi_{j+1} \in R_\Phi(m_0) \setminus LR_Z$  then  $t(\phi_1 \dots \phi_k) = \sigma_1 \dots \sigma_{j+1}$ , for some  $\sigma_1, \dots, \sigma_{j+1} \in \Sigma$  such that  $(m_0, \sigma_1 \dots \sigma_{j+1}) \in \text{dom } \|\phi_1 \dots \phi_{j+1}\|$ .*
- *If there is some  $j, 1 \leq j \leq k - 1$ , such that  $\phi_1 \dots \phi_j \in LR_Z$  and  $\phi_1 \dots \phi_{j+1} \notin R_\Phi(m_0)$  then  $t(\phi_1 \dots \phi_k) = \sigma_1 \dots \sigma_j$ , for some  $\sigma_1, \dots, \sigma_j \in \Sigma$  such that  $(m_0, \sigma_1 \dots \sigma_j) \in \text{dom } \|\phi_1 \dots \phi_j\|$ .*

**Proof.** Suppose  $\phi_1 \dots \phi_j \in LR_Z$ . Since  $\Phi$  is input-uniform, by the second rule of the definition it follows by induction on  $i, 1 \leq i \leq j$ , that there exist  $m_1, \dots, m_j \in M, \sigma_1, \dots, \sigma_j \in \Sigma, \gamma_1, \dots, \gamma_j \in \Gamma$  such that  $t(\phi_1 \dots \phi_j) = \sigma_1 \dots \sigma_j$  and  $\phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i), 1 \leq i \leq j$ .

Suppose  $\phi_1 \dots \phi_j \in LR_Z$  and  $\phi_1 \dots \phi_{j+1} \in R_\Phi(m_0) \setminus LR_Z$ . Since  $\Phi$  is input-uniform, there exists  $\sigma_{j+1} \in \Sigma$  such that  $(m_j, \sigma_{j+1}) \in \text{dom } \phi_{j+1}$ . Then, by the second rule of the definition,  $t(\phi_1 \dots \phi_{j+1}) = \sigma_1 \dots \sigma_{j+1}$ . Furthermore, by the fourth rule,  $t(\phi_1 \dots \phi_{j+i}) = t(\phi_1 \dots \phi_{j+1}) = \sigma_1 \dots \sigma_{j+1}, i \geq 1$ .

Suppose  $\phi_1 \dots \phi_j \in LR_Z$  and  $\phi_1 \dots \phi_{j+1} \notin R_\Phi(m_0)$ . Then, by the third rule of the definition,  $t(\phi_1 \dots \phi_{j+1}) = \sigma_1 \dots \sigma_j$ . Furthermore, by the fourth rule,  $t(\phi_1 \dots \phi_{j+i}) = t(\phi_1 \dots \phi_{j+1}) = \sigma_1 \dots \sigma_j, i \geq 1$ .  $\square$

The role of a test function is to test, using appropriate input symbols, if the realisable sequences of processing functions in  $Z$  have been implemented, hence the name. This idea is formalised by the following lemma.

**Lemma 10.2.** *Suppose  $\Phi$  is input-uniform and output-distinguishable. Let  $t : \Phi^* \rightarrow \Sigma^*$  be a test function of  $Z$  and  $Y \subseteq \Phi^*$ . If for all  $s \in t(Y), f_Z(s) = f_{Z'}(s)$  then  $LR_Z \cap Y = LR_{Z'} \cap Y$ .*

**Proof.** Let  $p \in Y$ . We prove that  $p \in LR_Z$  if and only if  $p \in LR_{Z'}$ .

Suppose  $p = \phi_1 \dots \phi_k \in LR_Z$ . Then  $t(p) = \sigma_1 \dots \sigma_k$  and there exist  $\gamma_1, \dots, \gamma_k \in \Gamma, m_1, \dots, m_k \in M$  such that  $\phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i), 1 \leq i \leq k$ . Since  $f_Z(\sigma_1 \dots \sigma_k) = f_{Z'}(\sigma_1 \dots \sigma_k)$ , 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  $\phi'_i(m'_{i-1}, \sigma_i) = (\gamma_i, m'_i), 1 \leq i \leq k$ , where  $m'_0 = m_0$ . Since  $\Phi$  is output-distinguishable, by induction on  $i, 1 \leq i \leq k$ , it follows that  $\phi'_i = \phi_i$  and  $m'_i = m_i$ . Thus  $p \in LR_{Z'}$ .

Suppose  $p = \phi_1 \dots \phi_k \in LR_{Z'}$ . We prove by contradiction that  $p \in LR_Z$ . Assume  $p \notin LR_Z$ . Let  $j, 0 \leq j \leq k - 1$ , be the largest integer for which  $p = \phi_1 \dots \phi_j \in LR_Z$ . Then  $\phi_1 \dots \phi_{j+1} \in R_\Phi(m_0) \setminus LR_Z$ , so  $t(p) = \sigma_1 \dots \sigma_{j+1}$  and there exist  $\gamma_1, \dots, \gamma_{j+1} \in \Gamma, m_1, \dots, m_{j+1} \in M$  such that  $\phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i), 1 \leq i \leq j + 1$ . Since  $f_Z(\sigma_1 \dots \sigma_{j+1}) = f_{Z'}(\sigma_1 \dots \sigma_{j+1})$  and  $\phi_1 \dots \phi_{j+1} \in LR_{Z'}$ , analogously to above, it follows that  $\phi_1 \dots \phi_{j+1} \in LR_Z$ , which is a contradiction.  $\square$

As  $Z$  and  $Z'$  are completely defined, the input sequence used to check the existence of a sequence of processing functions in the implementation will also check the existence of all its prefixes, as shown by the following lemma.

**Lemma 10.3.** *Suppose  $\Phi$  is input-uniform and output-distinguishable and  $Z$  and  $Z'$  are completely defined. Let  $t : \Phi^* \rightarrow \Sigma^*$  be a test function of  $Z$  and  $Y \subseteq \Phi^*$ . If for all  $s \in t(Y), f_Z(s) = f_{Z'}(s)$  then  $LR_Z \cap \text{pref}(Y) = LR_{Z'} \cap \text{pref}(Y)$ .*

**Proof.** Let  $p \in Y$  and  $p_1 \in \text{pref}(p)$ . Then  $t(p_1) \in \text{pref}(t(p))$ . Since  $Z$  and  $Z'$  are completely defined, from  $f_Z(t(p)) = f_{Z'}(t(p))$  it follows that  $f_Z(t(p_1)) = f_{Z'}(t(p_1))$ . Then the result follows from Lemma 10.2 since  $p$  and  $p_1$  have been arbitrarily chosen.  $\square$

Once the product machine  $P(Z, Z')$  has been defined and a mechanism for computing the values of a test function  $t$  is in place, a test suite can be constructed by first generating sequences of processing functions to check whether the

Fail state of  $A_{P(Z,Z')}$  is reachable and then transforming them into input sequences through  $t$ . The process is detailed in the following section.

## 11. Test suite generation

The first step in the construction of the test suite is the selection of two sets of sequences of processing functions,  $S_r$  and  $W_r$ , and of a relation  $d_r$  on the states of  $Z$  as follows:

- $S_r \subseteq LR_Z$  is a finite set of realisable sequences such that
  - $\varepsilon \in S_r$  and
  - no state in  $Z$  is reached by more than one sequence in  $S_r$ , i.e. for every two distinct sequences  $p_1, p_2 \in S_r$ ,  $F^*(q_0, p_1) \neq F^*(q_0, p_2)$ . $S_r$  will be used to reach  $r$ -reachable states in  $Z$ .
- $W_r \subseteq \Phi^*$  is a finite set of processing functions.  $W_r$  will be used to  $r$ -distinguish between  $r$ -distinguishable states of  $Z$ .  $W_r$  is required to be non-empty, so when no sequences are used to  $r$ -distinguish between states of  $Z$ , we will use  $W_r = \{\varepsilon\}$  instead of  $W_r = \emptyset$ .  $W$  will contain the empty sequence  $\varepsilon$ .
- $d_r : Q \longleftrightarrow Q$  is a relation on the states of  $Z$  that satisfies the following condition: for every two states  $q_1, q_2 \in Q$ , if  $(q_1, q_2) \in d_r$  then  $q_1$  and  $q_2$  are  $r$ -distinguished by  $W_r$ . The relation  $d_r$  identifies the pairs of states that are known to be  $r$ -distinguished by  $W_r$ . For simplicity,  $d_r$  is required to be symmetric.

Naturally, it is normally desirable that

- $S_r$  is an  $r$ -state cover of  $Z$ ,
- $W_r$  is an  $r$ -characterisation set of  $Z$  and
- all pairwise  $r$ -distinguishable states of  $Z$  are known to be  $r$ -distinguished by  $W_r$ , i.e.  $(q_1, q_2) \in d_r$  if and only if  $q_1$  and  $q_2$  are  $r$ -distinguishable, but these restrictions will not be introduced.

The set of all states reached by sequences in  $S_r$  is denoted by  $Q_r$ , i.e.  $Q_r = \{q \in Q \mid \text{there exists } p \in S_r \text{ such that } F^*(q_0, p) = q\}$ . As all sequences in  $S_r$  are realisable, all states in  $Q_r$  are  $r$ -reachable. Furthermore, since  $\varepsilon \in S_r$ , the initial state of  $Z$  is contained in  $Q_r$ .

Let  $Q_1, \dots, Q_j$  denote the maximal sets of states of  $Z$  that are known to be pairwise  $r$ -distinguished by  $W_r$ , i.e. for every  $q_1, q_2 \in Q_i$  and every  $q_3 \in Q \setminus Q_i$ ,  $(q_1, q_2) \in d_r$  and  $(q_1, q_3) \notin d_r$ ,  $1 \leq i \leq j$ . Let also  $Q'_i = Q_i \cap Q_r$ ,  $1 \leq i \leq j$ .

**Example 11.1.** Consider  $Z$  as defined in Example 4.1 and  $k = 3$ . As shown earlier, all states of  $Z$  are  $r$ -reachable and  $S_r = \{\varepsilon, popErr, pushSucc, pushSucc\}$  is an  $r$ -state cover of  $Z$ . Furthermore, all pairs of states, except  $(Popped, Loaded)$  and  $(Pushed, Loaded)$ , are  $r$ -distinguishable and  $W_r = \{errId, popErr, popSucc\}$  is an  $r$ -characterisation set of  $Z$ .

Suppose  $S_r$  and  $W_r$  are the chosen sets of sequences and every  $r$ -distinguishable pair of states is known to be  $r$ -distinguished by  $W_r$ , i.e.  $(q_1, q_2) \in d_r$  if and only if  $q_1$  and  $q_2$  are  $r$ -distinguishable. Then there are two maximal sets of states known to be pairwise  $r$ -distinguished by  $W_r$ :  $Q_1 = \{Error, Pushed, Popped\}$  and  $Q_2 = \{Error, Loaded\}$ . Since  $Q_r = Q$ ,  $Q'_1 = Q_1$  and  $Q'_2 = Q_2$ .

Given a state  $q \in Q_r$ , let  $p_q \in S_r$  denote the sequence in  $S_r$  that reaches  $q$ . As every state in  $Q_r$  is reached by exactly one sequence in  $S_r$ ,  $p_q$  is well defined. Suppose that a test function  $t : \Phi^* \rightarrow \Sigma^*$  has been defined for all sequences of processing functions in  $S_r$ .

Given a state  $q \in Q_r$ , the set  $V(q)$  is defined to consist of all sequences  $x \in \Phi^* \setminus \{\varepsilon\}$  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  $n' - \text{card}(Q'_i) + 1$  times when followed from  $q$  in  $A_Z$  (the initial state of the path is not included in the counting) and this condition does not hold for any proper prefix of  $x$ , i.e.
  - there exists  $i$ ,  $1 \leq i \leq j$ , such that  $\text{card}(\{F^*(q, y) \mid y \in \text{pref}(x) \setminus \{\varepsilon\}\}) = n' - \text{card}(Q'_i) + 1$  and
  - for all  $i$ ,  $1 \leq i \leq j$ , and all  $x_1 \in \text{pref}(x) \setminus \{x\}$ ,  $\text{card}(\{F^*(q, y) \mid y \in \text{pref}(x_1) \setminus \{\varepsilon\}\}) < n' - \text{card}(Q'_i) + 1$ .

Informally,  $V(q)$  is defined such that to contain only “minimal” paths of  $A_{\mathbb{P}(Z, Z')}$  that may reach the *Fail* state. Such a minimal path will not have visited the same 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  $S_r$ . If a path  $x$  visits states from some  $Q_i$ , a tester can use  $W_r$  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  $n_i$  distinct states will be visited in  $Z'$ . Thus,  $n_i$  cannot exceed the upper bound  $n'$  on the number of states of  $Z'$  plus one (for the *Fail* state). On the other hand, among the states of  $Q_i$  there are  $\text{card}(Q'_i)$  states that can be reached by sequences from  $S_r$ . As  $S_r$  will also reach the corresponding states of  $Z'$ , this will leave  $\text{card}(Q'_i)$  less pairs of states to explore. Thus,  $n_i \leq n' - \text{card}(Q'_i) + 1$ .

The set  $V(q)$  is finite and can be computed, as shown in what follows.

**Lemma 11.1.** *Each sequence in  $V(q)$  has length at most  $n \cdot n'$ .*

**Proof.** Suppose  $V(q)$  contains a sequence  $x$  of length greater than  $n \cdot n'$ . Let  $x_1$  be the prefix of  $x$  of length  $n \cdot n'$ . Then for every  $i$ ,  $x_1$  visits the states of  $Q_i$  at most  $n' - \text{card}(Q'_i)$  times. Since at least one of the sets  $Q'_i$  is not empty (it contains the initial state), the maximum number of states visited by  $x_1$  will be  $(n - 1)n' + n' - 1 = n \cdot n' - 1$ . Thus the length of  $x_1$  is at most  $n \cdot n' - 1$ , which is a contradiction.  $\square$

The set  $V(q)$  can be constructed by devising a tree in which each path  $x$  from the root  $q$  represents the tail-end part of a realisable sequence  $p_q x$ . A path meets the termination criterion when it visits states from some  $Q_i$  exactly  $n' - \text{card}(Q'_i) + 1$  times. In this case, the path need not be extended further, so the node will be a leaf. A formal description of the procedure is given below. Note that the procedure not only constructs  $V(q)$ , but also the values of a test function  $t$  for the sequences in  $\{p_q\}V(q)$ . It will transpire that these values are the actual input sequences used for testing. In what follows  $\pi_i : A_1 \times A_2 \rightarrow A_i$  is used to denote the projection on the  $i$ th component,  $1 \leq i \leq 2$ . For simplicity, it is assumed that  $t(p_q)$ ,  $m_q = \pi_2(\|p_q\|(m_0, t(p_q)))$  and the sets  $Q_1, \dots, Q_j$  and  $Q'_1, \dots, Q'_j$  have already been determined.

Input  $Z, n', q, p_q, t(p_q), m_q, Q_1, \dots, Q_j$  and  $\text{card}(Q'_1), \dots, \text{card}(Q'_j)$ ;  
 $n_1 := 0, \dots, n_j = 0$ ;  $X := \emptyset$ ;  $Y := \{((\varepsilon, \varepsilon), (q, m_q), (n_1, \dots, n_j))\}$ ;  
 Repeat

  For  $y$  in  $Y$  do

$Y := Y \setminus y$ ;  $((p, s), (q, m), (n_1, \dots, n_j)) := y$ ;

    For  $\phi$  in  $\Phi$  such that  $(q, \phi) \in \text{dom } F$  do

      Find  $\sigma \in \Sigma$  such that  $(m, \sigma) \in \text{dom } \phi$

      If such  $\sigma$  was found then

        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  $n'_i = n' - \text{card}(Q'_i) + 1$  then

$X = X \cup \{(p\phi, s\sigma)\}$

        Else  $Y = Y \cup \{((p\phi, s\sigma), (F(q, \phi), \pi_2(\phi(m, \sigma))), (n'_1, \dots, n'_j))\}$ ;

Until  $Y = \emptyset$ ;

$V = \emptyset$ ;  $TV = \emptyset$ ;

For  $x$  in  $X$  do

$V = V \cup \{\pi_1(x)\}$ ;  $TV = (p_q \pi_1(x), t(p_q) \pi_2(x))$ ;

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. Lemma 11.1 ensures that the number of iterations is finite. The algorithm outputs the set  $V(q)$  and the values of a test function  $t$  for sequences in  $\{p_q\}V(q)$ .

For  $Z$  as defined in Example 4.1,  $k = 3$ ,  $n' = 5$ ,  $q = \text{Popped}$ ,  $p_q = \varepsilon$ ,  $t(p_q) = \varepsilon$ ,  $m_q = \varepsilon$ ,  $j = 2$ ,  $Q_1 = Q'_1 = \{\text{Error}, \text{Pushed}, \text{Popped}\}$  and  $Q_2 = Q'_2 = \{\text{Error}, \text{Loaded}\}$ , the tree generated by the procedure is represented in



Fig. 4. The tree generated for  $V(\text{Popped})$ .

Fig. 4. Each node in the tree has a corresponding state of  $Z$  and two values indicating the number of times the path from the root to that node has encountered states from  $Q_1$  and  $Q_2$ , respectively (the corresponding memory value is not shown in order to keep the figure simple). The paths in the tree are sequences of processing functions; for simplicity, the input sequences used to drive these functions are omitted. The leaf nodes are in bold. A node is a leaf if the path from the root to it has encountered (after the root) three states that are contained in  $Q_1$  or four states that are contained in  $Q_2$ .

**Lemma 11.2.** For any choice of  $S_r$ ,  $W_r$  and  $d_r$  and for every  $q \in S_r$ , the set  $V(q)$  can be computed.

**Proof.** As the sets  $Q_r$  and  $Q_i$ ,  $1 \leq i \leq n$ , can be computed from  $S_r$  and  $d_r$ , respectively,  $V(q)$  can be computed using the procedure above.  $\square$

Once we have constructed the sets  $V(q)$ , a test suite can be generated by taking all sequences in  $\{p_q\}pref(V(q))$ , concatenating them with  $W_r$  and applying a test function  $t : \Phi^* \rightarrow \Sigma^*$  to every resulting sequence of processing functions.

As the sets  $V(q)$  are finite and computable, the set

$$U = \bigcup_{q \in Q_r} \{p_q\}pref(V(q))W$$

is also finite and computable.

**Lemma 11.3.** *For any choice of  $S_r$ ,  $W_r$  and  $d_r$ , the set  $U$  is finite and can be computed.*

**Proof.** Follows from Lemmas 11.1 and 11.2.  $\square$

The remainder of the section shows that  $t(U)$  is a test suite.

**Lemma 11.4.** *Let  $p_1, p_2 \in LR_Z$ ,  $q_1, q_2 \in Q$  such that  $F^*(q_0, p_1) = q_1$  and  $F^*(q_0, p_2) = q_2$ . Suppose  $W_r$   $r$ -distinguishes between  $q_1$  and  $q_2$  in  $Z$ . If for all  $s \in t(\{p_1, p_2\}W_r)$ ,  $f_Z(s) = f_{Z'}(s)$  then there exist  $q'_1, q'_2 \in Q'$  such that  $F'^*(q'_0, p_1) = q'_1$  and  $F'^*(q'_0, p_2) = q'_2$  and  $W_r$  distinguishes between  $q'_1$  and  $q'_2$  in  $A_{Z'}$ .*

**Proof.** As  $p_1, p_2 \in LR_Z$  and  $p_1, p_2 \in pref(\{p_1, p_2\}W_r)$ , by Lemma 10.3  $p_1, p_2 \in LR_{Z'}$ , so there exist such  $q'_1$  and  $q'_2$ .

Let  $m_1, m_2 \in M$ ,  $g_1, g_2 \in \Gamma^*$  such that  $\|p_1\|(m_0, t(p_1)) = (g_1, m_1)$  and  $\|p_2\|(m_0, t(p_2)) = (g_2, m_2)$ . Since  $W_r$   $r$ -distinguishes between  $q_1$  and  $q_2$  in  $Z$ ,  $LR_Z(q_1, m_1) \cap W_r \neq LR_Z(q_2, m_2) \cap W_r$ . Since for all  $s \in t(\{p_1, p_2\}W_r)$ ,  $f_Z(s) = f_{Z'}(s)$ , by Lemma 10.2,  $LR_Z(q_1, m_1) \cap W_r = LR_{Z'}(q'_1, m_1) \cap W_r$  and  $LR_Z(q_2, m_2) \cap W_r = LR_{Z'}(q'_2, m_2) \cap W_r$ . Thus  $LR_{Z'}(q'_1, m_1) \cap W_r \neq LR_{Z'}(q'_2, m_2) \cap W_r$ . Since  $Z'$  is controllable,  $L_{A_{Z'}}(q'_1) \cap W_r \neq L_{A_{Z'}}(q'_2) \cap W_r$ , so  $W_r$  distinguishes between  $q'_1$  and  $q'_2$  in  $A_{Z'}$ .  $\square$

**Lemma 11.5.** *Let  $q \in Q$  and  $x \in V(q)$ . Suppose for all  $s \in t(S_r W_r \cup \{p_q x\}W_r)$ ,  $f_Z(s) = f_{Z'}(s)$ . Then the path in  $A_{P(Z, Z')}$  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  $S_r$ .*

**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  $A_Z$ ,  $A_{Z'}$  and  $A_{P(Z, Z')}$ , respectively. The root states are not included when referring to these paths. First note that, by Lemmas 10.2 and 9.1, such paths also exist in  $A_{Z'}$  and  $A_{P(Z, Z')}$ .

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 11.4, since  $W_r$  pairwise  $r$ -distinguishes between the states in  $Q_i$ , it will also pairwise distinguish between the corresponding states in  $A_{Z'}$ . Thus  $path_{Z'}(x, p_q)$  will visit at least  $n' - card(Q'_i) + 1$  distinct states and the sequences in  $S_r$  will reach at least other  $card(Q'_i)$  states. This implies that  $Z'$  has more than  $n'$  states, which is a contradiction.  $\square$

**Lemma 11.6.** *Let  $A$  be an FA over input alphabet  $\Sigma$ . Suppose  $Y \subseteq \Sigma^*$  is a set of input sequences such that*

- $\varepsilon \in \Sigma^*$ ,
- every state of  $Z$  that is reached by a sequence in  $Y\Sigma$  has already been reached by some sequence in  $Y$ .

*Then every reachable state of  $A$  is reached by some sequence in  $Y$ .*

**Proof.** We provide a proof by contradiction. Let  $Q_Y$  be the set of states reached by sequences in  $Y$ . Suppose there exists a reachable state  $q$  of  $A$  such that  $q \notin Q_Y$ . Since  $\varepsilon \in Y$ , the initial state of  $A$  is contained in  $Q_Y$ , so  $q$  is reachable from the states in  $Q_Y$ . Let  $s = \sigma s'$ ,  $\sigma \in \Sigma$ ,  $s \in \Sigma^*$ , be the shortest input sequence that reaches  $q$  from a state in  $Q_Y$  and let  $q_1 \in Q_Y$  be that state (if more than one such sequence and state exist then the choice is arbitrary). Let also  $q_2$  be the state reached by  $\sigma$  from  $q_1$ . Then  $q_2$  can be reached by some sequence in  $Y\Sigma$ , so, by the lemma hypothesis,  $q_2 \in Q_Y$ . Thus  $s'$  is a sequence shorter than  $s$  that reaches  $q$  from a state in  $Q_Y$ . This provides a contradiction, as required.  $\square$

**Lemma 11.7.** *If for all  $s \in t(U)$ ,  $f_Z(s) = f_{Z'}(s)$  then the Fail state of  $A_{P(Z,Z')}$  is not reachable.*

**Proof.** Given  $q \in Q_r$ , let  $V_{pref}(q) = pref(V(q)) \setminus V(q)$ . Since  $\varepsilon \notin V(q)$ ,  $\varepsilon \in V_{pref}(q)$ . From the construction of  $V(q)$ , it follows that  $\{p_q\}V_{pref}(q)\Phi \cap LR_Z \subseteq \{p_q\}pref(V(q))$ . Let  $E = \bigcup_{q \in Q_r} \{p_q\}pref(V(q))$  and  $E_{pref} = \bigcup_{q \in Q_r} \{p_q\}V_{pref}(q)$ . Then  $E_{pref}\Phi \cap LR_Z \subseteq E$ .

We prove that  $E_{pref}\Phi \cap L_{A_{P(Z,Z')}} \subseteq E$ . Let  $p \in E_{pref}\Phi \cap L_{A_{P(Z,Z')}}$ . Since  $p \in L_{A_{P(Z,Z')}}$ , by Lemmas 9.1 and 9.2 either

1.  $p \in L_{AZ} \cap L_{AZ'}$  or
2.  $p \in L_{AZ'} \setminus L_{AZ}$  and  $p = p_1\phi$  for some  $p_1 \in L_{AZ} \cap L_{AZ'}$  and  $\phi \in \Phi$ .

Thus we have the following two cases:

1.  $p \in E_{pref}\Phi \cap L_{AZ} \cap L_{AZ'}$ . Since  $E_{pref}\Phi \cap L_{AZ} \cap L_{AZ'} \subseteq E_{pref}\Phi \cap L_{AZ} \cap R_\Phi(m_0) = E_{pref}\Phi \cap LR_Z \subseteq E$ , it follows that  $p \in E$ .
2.  $p \in L_{AZ'} \setminus L_{AZ}$  and  $p = p_1\phi$  for some  $p_1 \in E_{pref} \cap L_{AZ} \cap L_{AZ'}$  and  $\phi \in \Phi$ . We prove by contradiction that  $p \in L_{A_{P(Z,Z')}}$ . Assume  $p \notin L_{A_{P(Z,Z')}}$ . Since  $p \in R(m_0)$ , there exist  $s \in \Sigma^*$ ,  $\sigma \in \Sigma$ ,  $g \in \Gamma^*$ ,  $\gamma \in \Gamma$ ,  $m, m' \in M$  such that  $\|p_1\|(m_0, s) = (g, m)$  and  $\phi(m, \sigma) = (\gamma, m')$ . Since  $Z$  is completely defined and  $p \notin LR_Z$ , there exists  $\phi' \in \Phi$  with  $(m, \sigma) \in dom \phi'$  such that  $p_1\phi' \in LR_Z$ . Since  $dom \phi \cap dom \phi' \neq \emptyset$  and  $Z'$  is deterministic,  $p_1\phi' \notin LR_{Z'}$ . On the other hand,  $p_1\phi' \in E_{pref}\Phi \cap LR_Z$ , so  $p_1\phi' \in E$ . By Lemma 10.3,  $LR_Z \cap E = LR_{Z'} \cap E$ . Since  $E \subseteq LR_Z$ , it follows that  $E \subseteq LR_{Z'}$ . Thus  $p_1\phi' \in LR_{Z'}$ , which provides a contradiction, as required.

By Lemma 11.5, every state of  $A_{P(Z,Z')}$  that is reached by a sequence in  $E$  is also reached by some sequence in  $E_{pref}$ . As  $E_{pref}\Phi \cap L_{A_{P(Z,Z')}} \subseteq E$ , every state of  $A_{P(Z,Z')}$  that is reached by a sequence in  $E_{pref}\Phi$  has already been reached by some sequence in  $E_{pref}$ . Furthermore, since all sets  $V_{pref}(q)$  contain the empty sequence  $\varepsilon$ ,  $S_r \subseteq E_{pref}$ . Then, by Lemma 11.6,  $E_{pref}$  reaches all reachable states of  $A_{P(Z,Z')}$ . Thus, if the *Fail* state was reachable, then it would be reached by some sequence in  $E_{pref}$ .

On the other hand,  $E \subseteq LR_Z$ , so  $E_{pref} \subseteq L_{AZ}$ . Thus, by Lemma 9.2, the *Fail* state cannot be reached by any sequence in  $E_{pref}$ . Hence *Fail* is not reachable.  $\square$

**Theorem 11.1.** *The Fail state of  $A_{P(Z,Z')}$  is not reachable if and only if for all  $s \in t(U)$ ,  $f_Z(s) = f_{Z'}(s)$ .*

**Proof.** Follows from Lemmas 11.7 and 9.5.  $\square$

**Theorem 11.2.**  *$f_Z = f_{Z'}$  if and only if for all  $s \in t(U)$ ,  $f_Z(s) = f_{Z'}(s)$ .*

**Proof.** Follows from Theorem 11.1 and Lemma 9.5.  $\square$

Note that if all the states of  $Z$  are  $r$ -reachable and pairwise  $r$ -distinguishable,  $S_r$  is an  $r$ -state cover of  $Z$ ,  $W_r$  is an  $r$ -characterisation set of  $Z$  and all states of  $Z$  are known to be pairwise  $r$ -distinguished by  $W_r$ , then

$$U = (S_r\Phi[n' - n + 1] \cap LR_Z)W,$$

so the method reduces to an extension of the  $W$ -method to DSXMs. This particular case is a generalisation of the result given in [33], which extends the  $W$ -method only to *controllable* DSXM specifications.

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

## 12. Complexity

The final question that needs to be addressed is concerned with the size of the generated test suite and the complexity of the test generation algorithm. For  $U = S_r\Phi[n' - n + 1]W$ , as given by the application of the  $W$ -method to the associated finite automaton, the number of sequences in  $U$  is at most  $n^2 \cdot k^{n'-n+1}$  and the total length of all sequences in  $U$  is at most  $n^2 \cdot n' \cdot k^{n'-n+1}$ , where  $k = card(\Phi)$ . Typically, only a small fraction of the sequences in  $S_r\Phi[n' - n + 1]$

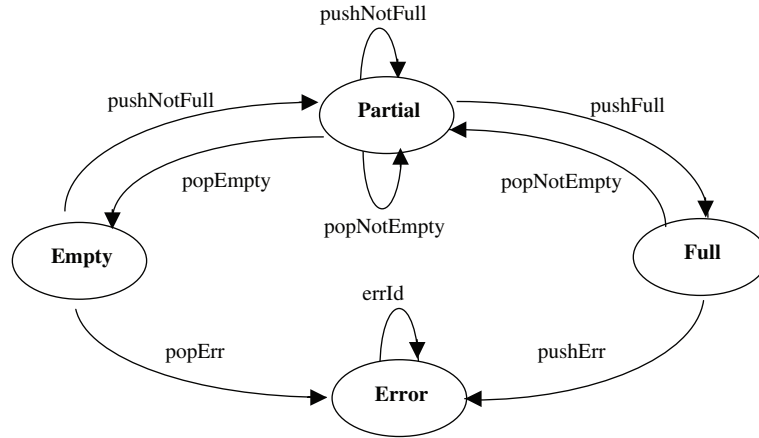


Fig. 5. The state-transition diagram of  $Z_{rev}$ .

are realisable, so the actual size of  $U = (S_r \Phi[n' - n + 1] \cap LR_Z)W$  is significantly lower. In the worst case, when  $S_r = W_r = \{\varepsilon\}$ , the upper bounds are proportional to  $k^{n' \cdot n}$ . However, this extreme is not normally encountered in practice. In usual applications, all states will be  $r$ -reachable and there will be (at most) only a few pairs of states that are not  $r$ -distinguishable.

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, in the case where all the states of  $Z$  are  $r$ -reachable and pairwise  $r$ -distinguishable,  $S_r$  is an  $r$ -state cover,  $W_r$  is an  $r$ -characterisation set of  $Z$  and all states of  $Z$  are known to be pairwise  $r$ -distinguishable by  $W_r$ , the complexity will be no more than  $C \cdot r \cdot n^2 \cdot n' \cdot k^{n' \cdot n + 1}$ , where  $r = \text{card}(\Sigma)$  and  $C$  is the maximum effort needed by a processing function to compute the next memory value, given the input and the current memory.

Clearly, the size of the test suite and the complexity of the test generation algorithm can be reduced by designing stream X-machines in which all states are pairwise  $r$ -distinguishable. Consider again the stream X-machine specification  $Z$  given in Example 4.1. For  $k \geq 4$ , there are no pairwise  $r$ -distinguishable states among *Popped*, *Loaded*, *Pushed*, so  $Z$  is not a suitable specification for deriving test suites. On the other hand, the system can be redesigned by considering states for empty, full and partial filled stack and splitting the *popSucc* and *pushSucc* functions accordingly, as shown in Fig. 5 (i.e. *popEmpty* produces an empty stack, whereas *popNotEmpty* produces a non-empty stack, etc.). It can be observed that the states of the revised stream X-machine specification  $Z_{rev}$  are pairwise  $r$ -distinguishable and that  $W_r = \{\text{errId}, \text{pushErr}, \text{popErr}\}$  is an  $r$ -characterisation set of  $Z_{rev}$ .

### 13. Test size reduction

From the above formulas, it can be observed that the size of the test suite depends exponentially on the difference between the upper bound  $n'$  on the number of states of  $Z'$  and the number  $n$  of states of  $Z$ . As  $Z'$  is assumed to be controllable, extremely large values of  $n'$  may sometimes be needed (in the worst case, the number of states of  $Z'$  may be proportional to the size of the memory). In such cases, the size of the test suite can be drastically reduced by taking into account an additional upper bound. Given  $Z'$  in the fault domain, we define  $v_{Z'}$  as the maximum number of states in  $Z'$  that have attainable memory values in common, i.e.  $v_{Z'} = \max_{m \in M} \text{card}\{q' \in Q' \mid m \in \text{MAtt}(q')\}$ . Typically, when the number of states  $n'_{Z'}$  of  $Z'$  is proportional to the size of the memory, the value of  $v_{Z'}$  is low. For  $Z'$  as defined in Example 7.1,  $n'_{Z'} = k + 2$ , while  $v_{Z'} = 2$ . An upper bound  $v$  on  $v_{Z'}$  can be used to prune the sequences in  $V(q)$ ,  $q \in Q_r$ , as explained in what follows.

- Consider the procedure for generating the set  $V(q)$ . Let  $x$  be a path from  $q$  that does not yet satisfy the termination criterion and will normally have to be extended. Suppose that there is a state  $q_x$  in the specification  $Z$  and a memory

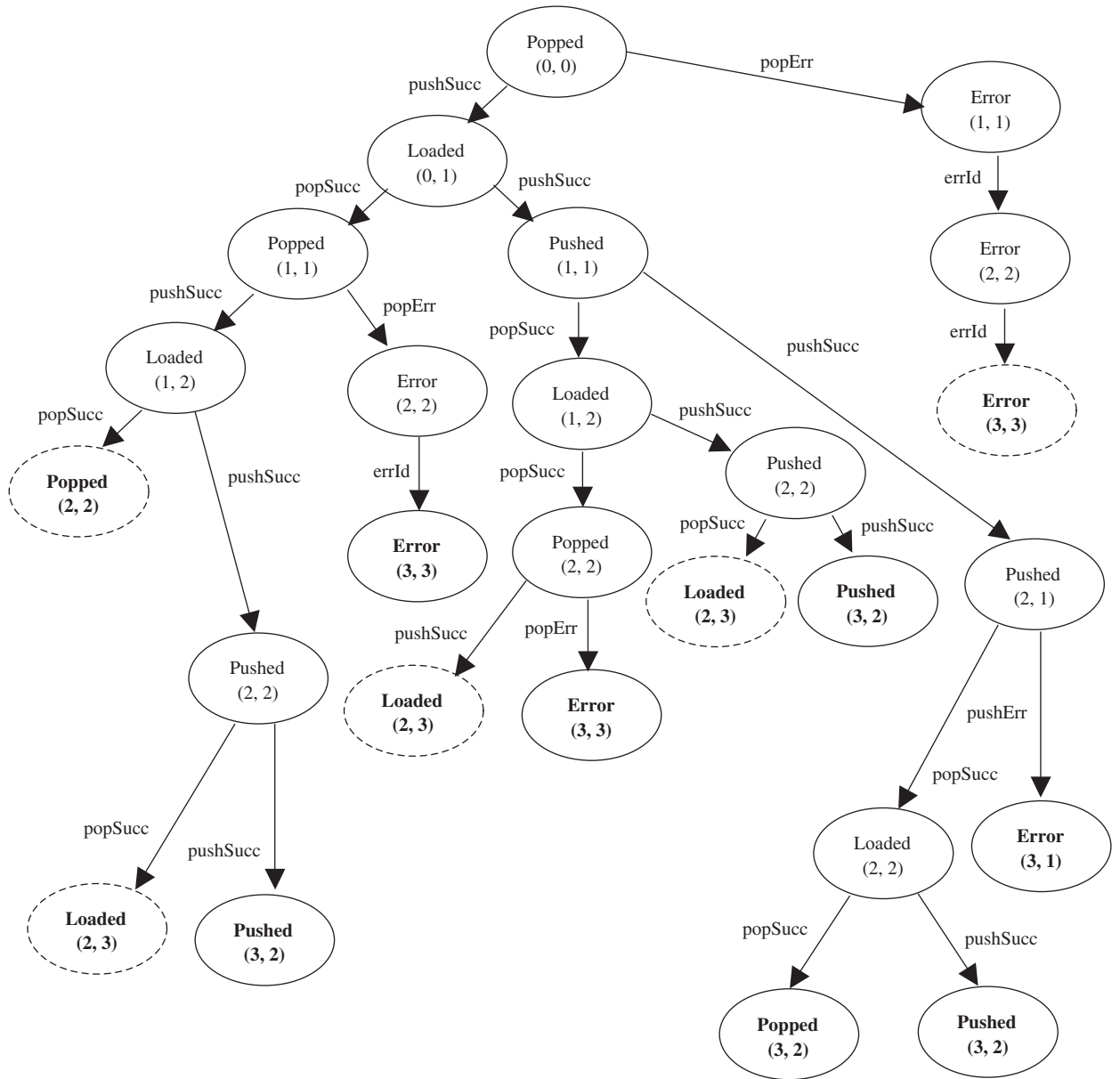


Fig. 6. The pruned tree for  $V(Popped)$ .

value  $m_x$  such that the application of  $x$  from  $q$  in  $Z$  has produced the memory value  $m_x$  for  $v + 1$  times when visiting the state  $q_x$ . As  $m_x$  may appear in at most  $v$  states of  $Z'$ , the path in  $A_{P(Z,Z')}$  formed by following  $x$  after  $p_q$  will contain a loop, so it need not be extended any further. Furthermore, since it has been established that the path in  $A_{P(Z,Z')}$  contains a loop without having to distinguish between the states of  $Z'$ , the sequence  $p_q x$  need not be concatenated with  $W_r$ . Thus, the test sequence can be constructed by simply applying a test function  $t$  to the sequence  $p_q x$ . For instance, for  $Z$  as defined in Example 4.1,  $q = Popped$ , and  $p_q = \epsilon$ , consider  $v = 2$ . Then the sequence  $x = pushSucc popSucc pushSucc popSucc$  need not be extended since it has visited the state  $q_x = Popped$  three times and each time has produced the memory value  $m_x = \epsilon$ .

- In the light of the above observation, the construction of the test function can be enhanced so that, each time a sequence  $x$  is extended with a processing function  $\phi$ , in the formula  $t(p_q x \phi) = t(p_q x)\sigma$ , the algorithm will select,

from the possible candidates, the input  $\sigma$  that yields the most frequent memory value produced by the path  $x$  when visiting the final state of  $p_q x \phi$ . Consider, in our example,  $q = Loaded$ ,  $p_q = pushSucc$ ,  $t(p_q) = e_1$  with  $e_1 \in E$  and the path  $x = pushSucc popSucc popSucc pushSucc$  from  $q$ . Then  $t(p_q x) = e_1 e_2 rem rem e_1$  for some  $e_2 \in E$ , so the path  $x$  from  $q$  has visited the state  $q_x = Loaded$  three times and each time has produced the memory value  $m_x = e_1$ . Thus  $x$  need not be extended.

Following these remarks, the tree for  $V(Popped)$  can be pruned as shown in Fig. 6. The leaf nodes resulting from the above pruning strategy are in dashed line.

## 14. Conclusions

This paper provides a strong generalisation of the existing stream X-machine based method, which removes from the specification the very restrictive controllability requirement and replaces it with input-uniformity. The new design for test conditions (output-distinguishability and input-uniformity) can be naturally introduced into a stream X-machine specification, without affecting the system functionality, so the generalised method can be applied to virtually any realistic stream X-machine specification.

As a consequence of removing the controllability requirement from the specification, the test generation algorithm uses a state-counting approach, rather than being based on Chow's  $W$ -method. The test suites generated using a state-counting strategy may be significantly larger than those given by the  $W$ -method, but, in practice, the extra effort spent for the application of a larger test suite is often exceeded by the effort associated with the process of introducing the extra functionality required by the controllability requirement and of removing this extra functionality after testing has been completed. Furthermore, in the case of large upper bounds on the number of states of the controllable implementation, the test size can be reduced by considering additional bounds.

Further work involves a similar generalisation for the *complete* stream X-machine based testing method [31], which enables the testing of the processing functions to be integrated into the testing of the overall system. The generalisation of the testing method based on *non-deterministic* stream X-machine specifications [36] may also be considered.

## Acknowledgements

The author would like to thank the anonymous reviewers, whose comments have improved the presentation of this paper.

## References

- [1] J. Aguado, T. Bălănescu, T. Cowling, M. Gheorghe, M. Holcombe, F. Ipatе, P Systems with replicated rewriting and stream X-machines (Eilenberg machines), *Fund. Inform.* 49 (1–3) (2002) 17–33.
- [2] T. Bălănescu, Generalized stream X machines with output delimited type, *Formal Aspects of Comput.* 12 (2000) 473–484.
- [3] T. Bălănescu, T. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, C. Vertan, Communicating stream X-machines are no more than X-machines, *J. Universal Comput. Sci.* 5 (1999) 494–507.
- [4] T. Bălănescu, M. Gheorghe, M. Holcombe, Deterministic stream X-machines based on grammar systems, in: C. Martin-Vide, V. Mitrană (Eds.), *Words, Sequences, Grammars, Languages: where Biology, Computer Science, Linguistics and Mathematics Meet*, Vol. 1, Kluwer, Dordrecht, 2000, pp. 13–23.
- [5] T. Bălănescu, M. Gheorghe, M. Holcombe, F. Ipatе, Testing collaborative agents defined as stream X-machines, *Advances in Artificial Life*, in: *Proc. Sixth European Conf. ECAL*, Prague, Czech Republic, 10–14 September, Springer, Berlin, 2001, pp. 296–305.
- [6] T. Bălănescu, M. Gheorghe, M. Holcombe, F. Ipatе, Eilenberg P Systems, in: Gh. Paun, G. Rozenberg, A. Salomaa, C. Zandron (Eds.), *Membrane Computing, International Workshop, WMC-CdeA 2002*, Curtea de Arges, Romania, August 2002, *Lecture Notes in Computer Science*, Vol. 2597, Springer, Berlin, 2003, pp. 43–57.
- [7] T. Bălănescu, M. Gheorghe, F. Ipatе, M. Holcombe, Formal black box testing for partially specified deterministic finite state machines, *Found. Comput. Decision Systems* 28 (1) (2003) 17–28.
- [8] T. Bălănescu, F. Ipatе, The Wp method for partially specified deterministic finite state machines, *Annals of Bucharest University, Computer Science*, Vol. LIII(1), 2004, pp. 47–60.
- [9] J. Barnard, J. Whitworth, M. Woodward, Communicating X-machines, *Inform. Software Tech.* 38 (1996) 401–407.
- [10] F. Bernardini, M. Gheorghe, M. Holcombe, P X systems = P systems + X machines, *Natural Comput.* 2 (3) (2003) 201–213.

- [11] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed, S. Vanak, Testing methods for X-machines, a review, *Formal Aspects of Comput.* (2006) to appear.
- [12] K.-T. Cheng, A.S. Krishnakumar, Automatic functional test generation using the extended finite state machine model, in: *Proc. 30th Design Automation Conference*, Dallas, Texas, USA, June 14–18, ACM Press, New Orleans, 1993, pp. 86–91.
- [13] T.S. Chow, Testing software design modelled by finite state machines, *IEEE Trans. Software Eng.* 4 (3) (1978) 178–187.
- [14] D. Cohen, *Introduction to Computer Theory*, second ed., Wiley, New York, 1991.
- [15] A. Cowling, H. Georgescu, C. Vertan, A structured way to use channels for communication in X-machine systems, *Formal Aspects of Comput.* 12 (6) (2000) 458–500.
- [16] J. Dick, A. Faivre, Automating the generation and sequencing of test cases from model-based specifications, *FME '93, First Internat. Symp. Formal Methods in Europe*, Odense, Denmark, April 1993, pp. 268–284, *Lecture Notes in Computer Science*, Vol. 670, Springer, Berlin.
- [17] S. Eilenberg, *Automata, Languages and Machines*, Vol. A, Academic Press, New York, 1974.
- [18] M. Fairtlough, M. Holcombe, F. Ipate, C. Jordan, G. Laycock, Z. Duan, Using an X-machine to model a video cassette recorder, *Current Issues in Electronic Modeling* 3 (1995) 141–161.
- [19] J.H. Fetzer, Program verification: the very idea, *Comm. ACM* 31 (1988) 1048–1063.
- [20] S. Fujiwara, G.v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, Test selection based on finite state models, *IEEE Trans. Software Eng.* 17 (6) (1991) 591–603.
- [21] M.C. Gaudel, Testing can be formal too, in: *TAPSOFT'95*, Springer, Berlin, March 1995, pp. 82–96.
- [22] H. Georgescu, C. Vertan, A new approach to communicating X-machines, *J. Universal Comput. Sci.* 6 (5) (2000) 490–502.
- [23] M. Gheorghie, Generalized stream X-machines and cooperating distributed grammar systems, *Formal Aspects of Comput.* 12 (6) (2001) 459–472.
- [24] R.M. Hierons, Testing from a Z specification, *J. Software Testing Verification and Reliability* 7 (1997) 19–33.
- [25] R.M. Hierons, M. Harman, Testing conformance to a quasi-non-deterministic stream X-machine, *Formal Aspects of Comput.* 12 (6) (2000) 423–442.
- [26] R.M. Hierons, M. Harman, Testing conformance of a deterministic implementation to a non-deterministic stream X-machine, *Theoret. Comput. Sci.* 323 (1–3) (2004) 191–233.
- [27] M. Holcombe, X-machines as a basis for dynamic system specification, *Software Eng. J.* 3 (1988) 69–76.
- [28] M. Holcombe, F. Ipate, *Correct Systems: Building a Business Process Solution*, Springer, Berlin, 1998.
- [29] M. Holcombe, F. Ipate, A., Grondoudis, Complete functional testing of safety-critical systems, *Proc. Second IFAC Workshop Safety and Reliability in Emerging Control Technologies*, Daytona Beach, Florida, USA, 1–3 November, Elsevier, Oxford, 1995, pp. 199–204.
- [30] F. Ipate, On the minimality of Stream X-machines, *Comput. J.* 46 (3) (2003) 295–306.
- [31] F. Ipate, Complete deterministic stream X-machine testing, *Formal Aspects of Comput.* 16 (4) (2004) 374–386.
- [32] F. Ipate, M. Holcombe, Another look at computability, *Informatica* 20 (1996) 359–372.
- [33] F. Ipate, M. Holcombe, An integration testing method that is proved to find all faults, *Internat. J. Comput. Math.* 63 (1997) 159–178.
- [34] F. Ipate, M. Holcombe, A method for refining and testing generalized machine specifications, *Internat. J. Comput. Math.* 68 (1998) 197–219.
- [35] F. Ipate, M. Holcombe, Specification and testing using generalized machines: a presentation and a case study, *Software Testing, Verification and Reliability* 8 (1998) 61–81.
- [36] F. Ipate, M. Holcombe, Generating test sequences from non-deterministic generalized stream X-machines, *Formal Aspects of Comput.* 12 (6) (2000) 443–458.
- [37] F. Ipate, M. Holcombe, An integrated refinement and testing method for stream X-machines, *Appl. Algebra Eng. Comm. Comput.* 13 (2) (2002) 67–91.
- [38] F. Ipate, M. Holcombe, Testing conditions for communicating stream X-machine systems, *Formal Aspects of Comput.* 13 (6) (2002) 431–446.
- [39] P. Kefalas, E. Kapeti, A design language and tool for X-machine specification, in: D.I. Fotadis, S.D. Nikolopoulos (Eds.), *Advances in Informatics*, World Scientific, Athens, 2000, pp. 134–145.
- [40] E. Kehris, G. Eleftherakis, P. Kefalas, Using X-machines to model and test discrete event simulation programs, in: N. Mastorakis (Ed.), *Systems and Control: Theory and Applications*, World Scientific and Engineering Society Press, Athens, 2000, pp. 163–171.
- [41] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines—a survey, *Proc. IEEE* 84 (8) (1996) 1090–1123.
- [42] T.J. Ostrand, M.J. Balcer, The category-partition method for specifying and generating functional tests, *Comm. ACM* 31 (6) (1989) 667–686.
- [43] A. Petrenko, N. Yevtushenko, G.V. Bochmann, Testing deterministic implementations from nondeterministic FSM specifications, *Proc. Ninth Internat. Workshop Testing of Communicating Systems (IWTC'S'96)*, 1996, pp. 125–140.
- [44] A.J.H. Simons, K. Bogdanov, M. Holcombe, Complete functional testing using Object Machines, Department of Computer Science Research Report CS-01-18, 2001.