Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Theoretical Computer Science 403 (2008) 176–191
Theoretical Computer Science
www.elsevier.com/locate/tcs

Testing data processing-oriented systems from stream X-machine models

Florentin Ipate[a,*], Mike Holcombe[b]

[a] *Department of Computer Science, University of Pitesti, Str Targu din Vale 1, 110040 Pitesti, Romania*
[b] *Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK*

Received 18 July 2006; received in revised form 15 October 2007; accepted 22 February 2008

Communicated by D. Sannella

Abstract

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 under test* (*IUT*), provided that the processing functions of the stream X-machine specification have been correctly implemented. The method was originally developed for *controllable* stream X-machines. A recent paper generalizes the original method by considering specifications that do not meet the controllability requirement. However, it is still required that a *controllable* stream X-machine model of the IUT exists and the size of the test suite produced strongly depends on the (estimated) upper bound on the number of states of this controllable model. While this assumption is in general reasonable for most interactive systems, it may produce unmanageable test suites for even simple data processing-oriented applications. This paper provides a new variant of the stream X-machine based testing method that no longer depends on the size of a *controllable* model of the IUT. In data processing-oriented applications, the new method can drastically reduce the size of the test suite produced at the expense of a (possibly) more complex generation process.
© 2008 Elsevier B.V. All rights reserved.

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

1. Introduction

Stream X-machines are a form of extended finite state machines that has received extensive study in recent years. A stream X-machine is a type of X-machine [13,18,19] 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 finite automaton (the *associated finite automaton*) in which the arcs are labelled by function names (the *processing functions*). Theoretical aspects, such as minimality [21], refinement [23,24] and communicating stream X-machine models [7,4,12,15,25] have been

* Corresponding author. Tel.: +40 21 4108964; fax: +40 248 216448.
E-mail addresses: fipate@ifsoft.ro (F. Ipate), m.holcombe@dcs.shef.ac.uk (M. Holcombe).

0304-3975/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.02.045

thoroughly investigated; a special interest has been given to the use of stream X-machines for simulating and verifying biology-inspired systems, such as P-systems [2,6,8].

As stream X-machines combine the dynamic features of finite state machines with data structures, they can naturally be used for system specification and development [19,14,31] and appropriate tools have been constructed [30]. Furthermore, 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 under test* (*IUT*), 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).

The testing method was initially developed for stream X-machine specifications that satisfy two design for test 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.

A recent paper [29] generalizes the original testing method by considering specifications that do not meet the controllability requirement. In the specification, this is replaced by a much laxer condition, called *input-uniformity*. However, the method still requires that the IUT can be modelled by a *controllable* stream X-machine and the size of the test suite produced strongly depends on the (estimated) upper bound on the number of states of this controllable model. While this assumption is in general reasonable for most interactive systems, it may produce unmanageable test suites for even simple data processing-oriented applications. In such applications, the control flow of the program is primarily determined by the values of the processed data. The user interaction with the system (if any) is typically limited to providing some of the data values to be processed (as inputs) and observing the outcome (the outputs). The system may also read values from data stores, such as data files or relational tables. For non-trivial data structures, the size of a controllable stream X-machine model of such a system is usually extremely large.

This paper provides a new variant of the stream X-machine based testing method that no longer depends on the size of a *controllable* model of the IUT. Furthermore, the specification used as basis for test generation may not satisfy the input-uniformity condition. Consequently, the new method can produce test suites of a considerably smaller size, in particular when applied to data processing-oriented systems. The downside is a (possible) more complex generation and evaluation process. However, in usual data processing-oriented applications, this drawback is more than compensated by the heavy reduction in the test size.

The paper is structured as follows. Section 2 introduces basic concepts of stream X-machines, Section 3 discusses the issue of reaching and identifying the states of (possibly non-controllable) stream X-machines, while Section 4 discusses the design test conditions required of the specification. The following five sections are dedicated to the new testing method: Section 5 identifies the fault model, Section 6 defines the product machine formed from the specification and the (unknown) model of the implementation, while in Section 7, sequences of processing functions are derived from the product machine through state counting; in the next section, these sequences are converted into input sequences through a mechanism called test function; the construction of the test suite is then assembled in Section 9, which also provides the results that validate this construction. The practical application of the method is discussed in Section 10. Finally, conclusions are drawn and further work is outlined in Section 11.

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. ϵ denotes the empty sequence. For a sequence $a \in A^*$, $|a|$ denotes the number of elements of a (in particular $|\epsilon|) = 0$). For $a, b \in A^*$, ab denotes the concatenation of sequences a and b. a^n is defined by $a^0 = \epsilon$ and $a^n = a^{n-1}a$, $n \geq 1$. For $U, V \subseteq A^*$, $UV = \{ab \mid a \in U, b \in V\}$; U^n is defined by $U^0 = \{\epsilon\}$ 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 $f : A \longrightarrow B$, dom f denotes the domain of f. For a finite set A, $card(A)$ denotes the number of elements in A.

2. 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*, that 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.

Definition 2.1. A *stream X-Machine* (abbreviated *SXM*) is a tuple $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, where:

- Σ is the finite *input alphabet*.
- Γ is the finite *output alphabet*.
- Q is the finite set of *states*.
- M is a (possibly infinite) set called *memory*.
- Φ is a finite set of distinct *processing functions*; a processing function is a non-empty (partial) function of type $M \times \Sigma \longrightarrow \Gamma \times M$.
- F is the (partial) *next-state function*, $F : Q \times \Phi \longrightarrow Q$.
- $q_0 \in Q$ 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 Φ. The automaton $A_Z = (\Phi, Q, F, q_0)$ over the alphabet Φ is called *the associated finite automaton* (abbreviated *associated FA*) of Z. A_Z is usually described by a state-transition diagram. The function F may be extended to take sequences from Φ^* to form the function $F^* : Q \times \Phi^* \longrightarrow Q$. $L_{A_Z}(q) = \{p \in \Phi^* \mid (q, p) \in \text{dom } F^*\}$ will denote the set of paths that can be traced out of state q. When $q = q_0$, this will be called the *language accepted* by Z and denoted L_{A_Z}.

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

To date stream X-machines have mainly been used to specify and test interactive systems [14,19,20,27]. However, by using "empty" or "silent" inputs, as in the next example, they can successfully model systems for which the user interaction is limited and the control flow is mainly determined by the processed data values.

Example 2.1. Consider a computer program that searches for the occurrence of a character $c \in CHAR$ in a string of characters $s \in CHAR^*$, both entered by the user.[1] The program returns the position of the first occurrence of c in s or 0 if c is not contained in s. After the search is complete, the user is given the option (represented as a two-valued set $OPT = \{yes, no\}$, disjoint from $CHAR^* \times CHAR$) to re-enter the character and repeat the search or to exit the program. The program can be modelled by a SXM Z with inputs $\Sigma = (CHAR^* \times CHAR) \cup OPT \cup \{\delta\}$, where $\delta \notin (CHAR^* \times CHAR) \cup OPT$ is the "empty" input associated with "silent" transitions[2] and outputs $\Gamma = N \cup \{null\}$, where N denotes the set of non-negative integers and $null \notin N$ is used when no visible output is produced. The memory $M = CHAR^* \times CHAR \times N$ will store s, c and a non-negative integer $i \in N$, used as a counter. The initial memory m_0 will not affect the program functionality and therefore can be chosen at random. The state-transition diagram of Z is as represented in Fig. 1, where 0 is the initial state, and the processing functions are defined as follows:
$initialize((s, c, i), (s', c')) = ((s', c', 1), null)$ if $(s', c') \in CHAR^* \times CHAR$,
$search((s, c, i), \delta) = ((s, c, i), null)$ if $i \leq |s|$,
$fail((s, c, i), \delta) = ((s, c, i), 0)$ if $i > |s|$,

[1] The string s is assumed to be bounded; for simplicity, the bound is not explicitly stated here, but will be referred to in later discussion.

[2] By abuse, we consider δ to be an input symbol; in testing, this special input can be simulated, for example, by inserting read operations or breakpoints into the original program; these will be removed after testing has been completed.

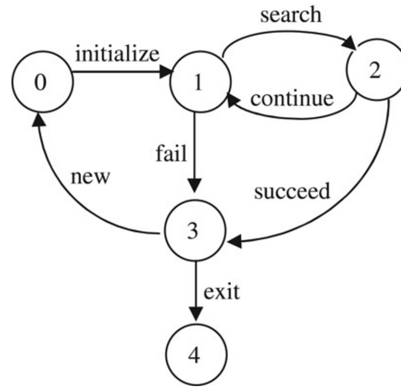

Fig. 1. The state-transition diagram of Z.

$continue((s, c, i), \delta) = ((s, c, i + 1), null)$ if $s[i] \neq c$, $1 \leq i \leq |s|$, where $s[i]$ denotes the ith element of s,
$succeed((s, c, i), \delta) = ((s, c, i), i)$ if $s[i] = c$, $1 \leq i \leq |s|$,
$new((s, c, i), yes) = ((s, c, i), null)$,
$exit((s, c, i), no) = ((s, c, i), null)$.

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 2.2. Given $p \in \Phi^*$, $\|p\| : M \times \Sigma^* \longrightarrow \Gamma^* \times M$ is defined by:

- $\|\epsilon\|(m, \epsilon) = (\epsilon, m)$, $m \in M$,
- 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 (function) computed* by the Z.

Definition 2.3. The relation computed by f, $f_Z : \Sigma^* \longleftrightarrow \Gamma^*$ 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 $\|p\|(m_0, s) = (g, m)$.

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 2.4. A SXM Z is said to be completely-defined if dom $f_Z = \Sigma^*$.

A SXM may be transformed into one that is completely-defined by assuming that the "refused" inputs produce a designated error output, which is not in the output alphabet of Z; this behaviour can be represented as self-looping transitions or transitions to an extra (error) state. In our example, the erroneous behaviour can be represented by three additional processing functions, $error_{data}$, $error_{opt}$ and $error_{\delta}$ that take inputs from $CHAR^* \times CHAR$, OPT and $\{\delta\}$, respectively, which will label appropriate self-looping transitions. That is, the state-transition diagram of the completely-defined SXM will contain the following (extra) self-looping transitions:

- $error_{opt}$ and $error_{\delta}$ in state 0,
- $error_{data}$ and $error_{opt}$ in states 1 and 2,
- $error_{data}$ and $error_{\delta}$ in state 3,
- $error_{data}$, $error_{opt}$ and $error_{\delta}$ in state 4.

For simplicity, the erroneous transitions are not represented in Fig. 1, but will be taken into account in future references to the example.

A *deterministic SXM* (abbreviated *DSXM*) is one in which there is at most one possible transition for any triplet (state, memory, input).

Definition 2.5. A SXM Z is said to be deterministic if 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 DSXM will compute a function f_Z rather than a relation. It can be observed that Z in our example is deterministic.

In this paper we only consider deterministic systems. As the implementation is expected to conform to the specification and the type of conformance considered is functional equivalence, the specification will also be deterministic. Consequently, in the remainder of the paper we will only refer to deterministic SXMs. Furthermore, without loss of generality, as discussed above, the specification will be assumed to be completely-defined.

3. Reaching and distinguishing states in a DSXM

This section is concerned with defining appropriate sequences of processing functions for reaching and distinguishing the states of a deterministic stream X-machine. The concepts of realisable and r-reachable states are originally defined in [29]. Additionally, this paper introduces the concept of separable states.

3.1. Realisable sequences

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 (see [29] for an example).

In order to determine which states can actually be reached or distinguished, we need to establish which sequences of processing functions in the associated automaton can be driven by input sequences from each state q and memory value m. Such sequences of processing functions are called *realisable* in q and m or simply realisable when $q = q_0$ and $m = m_0$. The sets of all these sequences are denoted by $LR_Z(q, m)$ and LR_Z, respectively. More formally, we have the following definition:

Definition 3.1. The set $R_\Phi(m) \subseteq \Phi^*$ is defined to consist of all sequences of processing functions $p = \phi_1 \ldots \phi_n \in \Phi^*$, $n \geq 0$, for which there exists $s = \sigma_1 \ldots \sigma_n \in \Sigma^*$ such that $(m, s) \in \text{dom } \|p\|$. Then $LR(q, m) = L_{A_Z}(q) \cap R_\Phi(m)$ and $LR = L_{A_Z} \cap R_\Phi(m_0)$.

Definition 3.2. Z is said to be *controllable* if all paths in A_Z are realisable, i.e. $LR_Z = L_{A_Z}$.

3.2. r-reachable states

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

Definition 3.3. State q of Z is said to be r-reachable if there exists $p \in LR_Z$ such that $F^*(q_0, p) = q$.

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

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

Definition 3.4. A set $S_r \subseteq LR_Z$ is called an r-state cover of Z if:

- $\epsilon \in S$.
- 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 $p_1, p_2 \in S_r$, $F^*(q_0, p_1) \neq F^*(q_0, p_2)$.

In our example, all states of Z are r-reachable and $S_r = \{\epsilon, initialize, initialize\ search, initialize\ fail, initialize\ fail\ exit\}$ is an r-state cover of Z.

3.3. Separable states

In [29], states in the specification are distinguished by applying a finite set of realisable sequences of processing functions to their current memory values. More precisely, the set $MAtt(q)$ of *attainable memory* values in state q is defined to consist of all memory values computed along all sequences in LR_Z that reach q, i.e. $m \in MAtt(q)$ if there exist $p \in LR$, $s \in \Sigma^*$ and $g \in \Gamma^*$ such that $F^*(q_0, p) = q$ and $\|p\|(m_0, s) = (g, m)$. Then, states q_1 and q_2 are said to be *r-distinguishable* if there exists a *finite* set of sequences Y such that for every $m_1 \in MAtt(q_1)$ and every $m_2 \in MAtt(q_2)$, $LR(q_1, m_1) \cap Y \neq LR(q_2, m_2) \cap Y$. The set Y is said to r-distinguish between q_1 and q_2. As shown in [29], 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 and, furthermore, even if such a set exists, it may not be finite.

In this paper, we will use a stronger condition, called separability, which requires states to be r-distinguished by sequences with overlapping domains.

Definition 3.5. States q_1 and q_2 are said to be *separable* if there exists a *finite* set of sequences Y such that for every $m_1 \in MAtt(q_1)$ and every $m_2 \in MAtt(q_2)$, there exist $p_1 \in LR(q_1, m_1) \cap Y$ and $p_2 \in LR(q_2, m_2) \cap Y$ such that $\text{dom}\ p_1 \cap \text{dom}\ p_2 \neq \emptyset$. Y is said to *separate* between q_1 and q_2.

Note that, since Z is deterministic, $p_2 \notin LR(q_1, m_1)$ and $p_1 \notin LR(q_2, m_2)$. Therefore, pairwise separable states are also pairwise r-distinguishable. The basic idea is that states that are separable in the specification will always be implemented as distinct states (otherwise the implementation will exhibit non-deterministic behaviour). Consequently, a set of processing sequences that separates states in the specification can also be used to distinguish between their corresponding states in the IUT.[3]

Definition 3.6. A *separating set* $W_s \subseteq \Phi^*$ of Z is a set of sequences of processing functions that separates between every pair of separable states of Z.

In our example, it can be observed that inputs from $CHAR^* \times CHAR$ will trigger the processing function *initialize* only when applied to state 0; in all the other states, they will produce the erroneous transition $error_{data}$. Thus $\{initialize,\ error_{data}\}$ separates state 0 from any other state. Similarly, $\{new,\ error_{opt}\}$ separates state 3 from any other state. In state 1, the system will perform one of the "silent" moves *search* or *fail*, whereas in any other state except 2, the empty input δ will trigger the erroneous transition $error_\delta$. Thus $\{search,\ fail,\ error_\delta\}$ separates between state 1 and any of 0, 3 and 4. Similarly, $\{continue,\ succeed,\ error_\delta\}$ separates between state 2 and any of 0, 3 and 4. On the other hand, states 1 and 2 are not separable since, for $i > |s|$, *fail* can be applied in 1 but there is no processing function whose domain overlaps with dom *fail* that can be applied in 2. Thus any pair of states except $(1, 2)$ is separable and $W_s = \{initialize,\ new,\ search,\ fail,\ continue,\ succeed, error_{data},\ error_{opt},\ error_\delta\}$ is a separating set of Z.

4. 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 Φ of the specification. The method provided in [29] requires the specification to satisfy two design for test conditions: output-distinguishability and input-uniformity.

Φ is *output-distinguishable* when the output produced in response to any given input determines which processing function has been applied.

Definition 4.1. Z is said to be 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 $\phi_1 = \phi_2$.

---

[3] When the DSXM model of implementation is controllable, like in [29], it is sufficient for the states of the specification to be r-distinguishable; otherwise, the stronger separability condition is required by Lemma 9.1.

This property allows the tester to determine the sequence of processing functions applied by examining the output sequence produced when given an input sequence.

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/or splitting some processing functions into two or more parts, in order to remove the overlapping of identical behaviour. In our example, it is not possible to distinguish between *search* and *continue*, but this can be address, for example, by displaying the value of the counter i. As it is reasonable to assume that the erroneous transitions can be distinguished from the normal behaviour, this enlargement of the output set is sufficient to transform Φ into an output-distinguishable type.

Informally, Φ is *input-uniform* [29] if 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. That is, if for a realisable sequence $\phi_1 \dots \phi_k$ of processing functions we have selected an input sequence $\sigma_1 \dots \sigma_{k-1}$ that drives $\phi_1 \dots \phi_{k-1}$, there will exist an input σ_k such that $\sigma_1 \dots \sigma_k$ drives $\phi_1 \dots \phi_k$. One particular case, considered in most publications addressing stream X-machine based testing [3,5,9,16,17,19,20,22–28] is when, for every processing function ϕ and every memory value m, there is an input σ that drives ϕ, i.e. $(m, \sigma) \in \text{dom } \phi$. Such a Φ is called *input-complete*. Clearly, a SXM having an input-complete type is controllable.

Input-uniformity can be achieved by designing a sufficient number of processing functions so that all inputs that trigger a functions will process "uniformly" any memory value (in the worst case, we will have processing functions triggered by single input sets). In our example, suppose that strings s of length between 0 and L are allowed. Then, in order to transform Φ into one that is input-uniform, we need to split *initialize* into $2L + 1$ initialization functions: one for each position i, $1 \leq i \leq L$, produced by a successful search and one for each length of s, $0 \leq | s | \leq L$, in the case of an unsuccessful search. For reasonably large L, the specification will be unmanageable, so the method will become impractical. An alternative will be to transform Φ into one that is input-complete by creating special inputs to trigger *search*, *fail*, *continue* and *succeed*. However, the addition and removal of these special inputs may prove difficult to implement, especially since these functions are actually driven by no input at all (they process the empty symbol δ).

The method presented in this paper will only require the specification to satisfy the output-distinguishability condition. Obviously, when Φ is not input-uniform, the process of converting sequences of processing functions into input sequences will increase in complexity but, in data processing-oriented applications, this increase may be more than compensated by the reduction in the size of the specification.

5. 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. Furthermore, when testing from a DSXM, it is normally assumed that Z and Z' have the same sets of processing functions (type) [3,5,9,16,17,19,20,23–28]. This is a consequence of the *reductionist* design and testing philosophy associated with stream X-machines [19]. Under this philosophy, it is assumed that the system is built from a set of trusted components,[4] the communication between them being modelled by the memory of the DSXM. These components will have been tested in a previous phase, such as unit testing, using DSXM based methods if they are expressible as the computations of other, simpler X-machines or using other functional testing approaches, such as category partition [35], if they carry out simple tasks on data structures (i.e. inserting and removing items from registers, stacks, files, etc.). Furthermore, if the components are imported from a library with a long history of successful use, their individual testing could safely be assumed done. System development will proceed through a sequence of steps, each of which involves building larger components from smaller components that have already been developed. For each such step, the testing problem reduces to checking that these components have been integrated in

[4] The word "component" is used here in its broad sense, of an object that adheres to a specification, e.g. a stream X-machine.

the correct way.[5] Naturally, the reliance on trusted components fits well with the increasing use of component based software development methodologies.[6]

Additionally, in [29], it is assumed that the IUT can be modelled by a *controllable* DSXM Z'; the (estimated) upper bound n' on the number of states of Z is then used as an input by the test generation procedure. In our example, such an upper bound will be proportional to the maximum length of the string s and, consequently, for even moderately long strings, the method given in [29] will become impractical. In what follows, we provide a method that, under well defined conditions, can generate a test suite without the need to estimate the size of a *controllable* model of the implementation. We will still assume that the upper bound n' on the number of states of Z' can be estimated, but the DSXM model Z' considered may not be controllable. Analogously to [29], the upper bound n' used is greater than or equal to the number of states n of the specification.

Thus, the fault model considered in this paper consists of all DSXMs Z' with the same type Φ as the specification for which the number of states is bounded by an estimated integer $n' \geq n$.

6. Cross-product automaton

Suppose $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ is the DSXM specification and $Z' = (\Sigma, \Gamma, Q', M, \Phi, F', q'_0, m_0)$ is the (unknown) DSXM model of the IUT. When type Φ is output-distinguishable, testing that Z' is functionally equivalent to Z corresponds to checking that every realisable sequence of processing functions in Z is also realisable in Z'.

On the other hand, 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 FA can follow. By adding to the cross-product FA an extra state, *Fail*, and transitions $F_P((q, q'), \phi) = \mathit{Fail}$ to correspond to transitions which can be taken by Z but not by Z', checking that every path in A_Z is also accepted by $A_{Z'}$ will correspond to checking the cross-product FA, denoted $A_{P(Z,Z')}$, in order to establish if the *Fail* state is reachable.

Definition 6.1. The *cross-product FA* formed from A_Z and $A_{Z'}$ is a finite automaton $A_{P(Z,Z')} = (\Phi, Q_P, F_P, (q_0, q'_0))$ in which $Q_P = (Q \times Q') \cup \{\mathit{Fail}\}$, $\mathit{Fail} \notin Q \times Q'$, 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 $(q, \phi) \in \text{dom } F$ and $(q', \phi) \notin \text{dom } F'$ then $F_P((q, q'), \phi) = \mathit{Fail}$.
  - Else $F_P((q, q'), \phi)$ is undefined.
- For $\phi \in \Phi$, $F_P(\mathit{Fail}, \phi)$ is undefined.

Since Φ is output-distinguishable, $f_Z = f_{Z'}$ if and only if $LR_Z = LR_{Z'}$; furthermore, since Z and Z' are completely-defined, $LR_Z = LR_{Z'}$ if and only if $LR_Z \subseteq LR_{Z'}$ (see [29] for proofs of these results). By combining these two observations, we obtain the following result:

Lemma 6.1. *Suppose Φ is output-distinguishable and Z and Z' are completely-defined. $f_Z = f_{Z'}$ if and only if there is no sequence from LR_Z that reaches Fail in $A_{P(Z,Z')}$.*

Proof. From the above observations it follows that $f_Z = f_{Z'}$ if and only $LR_Z \subseteq LR_{Z'}$. On the other hand, the *Fail* state can only be reached by sequences from $L_{A_Z} \setminus L_{A_{Z'}}$. Therefore, there is no sequence from LR_Z that reaches *Fail* if and only if $LR_Z \cap (L_{A_Z} \setminus L_{A_{Z'}}) = \emptyset$. Thus the result follows since $LR_Z \cap (L_{A_Z} \setminus L_{A_{Z'}}) = LR_Z \setminus LR_{Z'}$. □

5 Recent results also show that the generation of tests for the components of a DSXM Z can be integrated into the process of generating a test suite from Z [22].

6 It is worth noting that, unlike other extended finite state machine based approaches [33], stream X-machine based techniques do not involve the construction of an equivalent finite state machine (whose states are the state/memory pairs of the stream X-machine). Consequently, they avoid the state explosion associated with expanding out the memory and hence produce significantly reduced test suites. This is a direct consequence of this reductionist philosophy.

7. Deriving test sequences through state-counting

As explained in the previous section, establishing whether Z and Z' are functionally equivalent reduces to checking the reachability of the *Fail* state of the cross-product automaton. As $A_{Z'}$ is not known, we will need to construct a set of test sequences that, whenever *Fail* is reachable, will contain at least one sequence that reaches *Fail*. On the other hand, it will be sufficient to only consider the "minimal" (in a sense that will be explained later) paths that may reach the *Fail* state. The set of these "minimal" paths can be derived through state-counting [36]. The procedure is detailed in what follows:

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_s, and of a relation d_s on the states of Z as follows:

- $S_r \subseteq LR_Z$ is a finite set of realisable sequences such that $\epsilon \in S_r$ and no state in Z is reached by more than one sequence in S_r. S_r will be used to *reach* r-reachable states in Z.
- $W_s \subseteq \Phi^*$ is a finite set that will be used to *separate* between separable states of Z. W_s is required to be non-empty, so when no sequences are used to separate between states of Z, we will use $W_s = \{\epsilon\}$ instead of $W_s = \emptyset$.
- $d_s : Q \longleftrightarrow Q$ is a relation on the states of Z that satisfies the following condition: for every two states $q_1, q_2 \in Q$, if $(q_1, q_2) \in d_s$ then q_1 and q_2 are separated by W_s. The relation d_s identifies the pairs of states that *are known* to be separated by W_s. For simplicity, d_s is required to be symmetric.

Naturally, it is normally desirable that

- S_r is an r-state cover of Z,
- W_s is a separating set of Z and
- all pairwise separable states of Z are known to be separated by W_s, i.e. $(q_1, q_2) \in d_s$ if and only if q_1 and q_2 are separable.

but these restrictions will not be introduced.

The set of all states reached by sequences in S_r is denoted by Q_r. As all sequences in S_r are realisable, all states in Q_r are r-reachable. Furthermore, since $\epsilon \in S_r$, 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 separated by W_s, i.e. for every $q_1, q_2 \in Q_i$ and every $q_3 \in Q \setminus Q_i$, $(q_1, q_2) \in d_s$ and $(q_1, q_3) \notin d_s$, $1 \leq i \leq j$. Let also $Q'_i = Q_i \cap Q_r$, $1 \leq i \leq j$.

Suppose for Z in our example we have selected the r-state cover $S_r = \{\epsilon, \mathit{initialize}, \mathit{initialize\ search}, \mathit{initialize\ fail}, \mathit{initialize\ fail\ exit}\}$ and the separating set $W_s = \{\mathit{initialize}, \mathit{new}, \mathit{search}, \mathit{continue}, \mathit{succeed}, \mathit{fail}, \mathit{error}_{data}, \mathit{error}_{opt}, \mathit{error}_{\delta}\}$ and, furthermore, every separable pair of states is known to be separated by W_s, i.e. $(q_1, q_2) \in d_s$ if and only if q_1 and q_2 are separable. Then there will be two maximal sets of states known to be pairwise separable by $W_s : Q_1 = \{0, 1, 3, 4\}$ and $Q_2 = \{0, 2, 3, 4\}$. 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. Then the set $V(q)$ is defined to consist of all sequences $x \in \Phi^* \setminus \{\epsilon\}$ for which

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

Informally, $V(q)$ is defined to contain the "minimal" paths of the cross-product FA that may reach the *Fail* state. Such a minimal path will not have visited the same pair of states $(q, q') \in Q \times Q'$ twice and, furthermore, cannot contain pairs of states that have already been reached by sequences in S_r. If a path x visits states from some Q_i, a tester can use W_s 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,

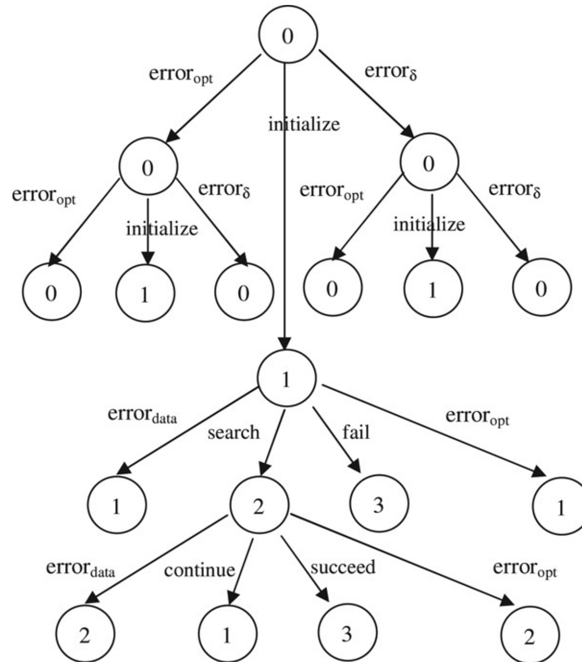

Fig. 2. The tree generated for $V(0)$.

among the states of Q_i there are $card(Q'_i)$ states that can be reached by sequences from S_r. As S_r will also reach the corresponding states of Z', this will leave $card(Q'_i)$ less pairs of states to explore. Thus, $n_i \leq n' - card(Q'_i) + 1$.

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 sequence $p_q\ x \in L_{A_Z}$. A path meets the termination criterion when it visits states from some Q_i exactly $n' - card(Q'_i) + 1$ times. In this case, the path need not be extended further, so the node will be a leaf.

For Z as in our example, $n' = 5$, $q = 0$, $p_q = \epsilon$, $Q_1 = Q'_1 = \{0, 1, 3, 4\}$ and $Q_2 = Q'_2 = \{0, 2, 3, 4\}$, the tree associated with $V(0)$ is represented in Fig. 2. The nodes of the tree are labelled by states and the arcs by processing functions. A node is a leaf if the path from the root to it has encountered (after the root) 2 states that are contained in Q_1 or 2 states that are contained in Q_2.

The above construction of the sets $V(q)$ ensures that any minimal path that may reach *Fail* is contained in the set

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

Then it will be sufficient to test each such path concatenated with all the elements of the distinguishing set W_s.

8. Test function

Suppose we have constructed the appropriate sequences of processing functions. We will then need a mechanism, called a test function, that translates sequences of processing functions into sequences of inputs. The concept of test function was originally defined for stream X-machines with Φ input-complete [26]. In [29], the definition was extended to the case in which Φ is input-uniform. We generalize this concept for any kind of stream X-machine.

Definition 8.1. A *test function* of a SXM Z is a function $t : \Phi^* \longrightarrow \Sigma^*$ that satisfies the following conditions:

- $t(\epsilon) = \epsilon$. (1)
- Let $p = \phi_1 \ldots \phi_k \in \Phi^*$, $k \geq 1$.
  - Suppose $\phi_1 \ldots \phi_{k-1} \in L_{A_Z}$ and there exist $\sigma_1, \ldots, \sigma_k \in \Sigma$, $\gamma_1, \ldots, \gamma_k \in \Sigma$ and $m_1, \ldots, m_k$ such that $\phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i)$, $1 \leq i \leq k$. Then $t(p) = \sigma_1 \ldots \sigma_k$ for some $\sigma_1, \ldots, \sigma_k$ that satisfy this condition; (2)
  - Otherwise, $t(p) = t(\phi_1 \ldots \phi_{k-1})$. (3)

The first rule states that the empty path is transformed into the empty input sequence, while the remaining two rules explain how $t(p)$ is defined for a non-empty sequence $p = \phi_1 \dots \phi_k$. The second rule states that if the longest proper prefix $\phi_1 \dots \phi_{k-1}$ is a path in A_Z then $t(p)$ will be an input sequence, if it exists, that drives p. Finally, when $\phi_1 \dots \phi_{k-1}$ is not a path in A_Z or no such input sequence exists, the construction of $t(p)$ is reduced, recursively, to the construction of $t(\phi_1 \dots \phi_{k-1})$.

In our example consider p_1 = *initialize search succeed*, $p_2 = p_1$ *initialize* and $p_3 = p_2$ *search succeed*. As $p_1 \in L_{A_Z}$, $t(p_1)$ and $t(p_2)$ will be input sequences that drive the corresponding paths. Thus, we can take $t(p_1) = (''a'','a')\ \delta\ \delta$ and $t(p_2) = (''a'','a')\ \delta\ \delta\ (''a'','a')$. As $p_2 \notin L_{A_Z}$, by applying twice the third rule we get $t(p_3) = t(p_2\ search) = t(p_2)$. On the other hand, if only non-empty strings s are allowed, there will be no input sequence to drive *initialize fail*, so the construction of t(*initialize fail*) reduces to the construction of t(*initialize*). Thus t(*initialize fail*) $= (''a'','a')$.

The role of a test function is to produce input sequences that test the implementation of the corresponding sequences of processing functions (hence the name). If $p = \phi_1 \dots \phi_k$ is a path in A_Z then $t(p)$ will be a sequence of inputs that drives p (if it exists). As Φ is output-distinguishable, if the application of this sequence to the implementation produces the specified outputs, it will ensure that p has been correctly implemented. On the other hand, if $p = \phi_1 \dots \phi_k$ is not a path in A_Z, then it is sufficient to extract the longest prefix $p = \phi_1 \dots \phi_i$ of p that is a path in A_Z and generate an input sequence that will check the existence of the path $p = \phi_1 \dots \phi_{i+1}$ in the implementation. If the application of this sequence produces the specified outputs, then we can deduce that the path $p = \phi_1 \dots \phi_{i+1}$ has not been implemented, and consequently, neither the longer path $p = \phi_1 \dots \phi_k$ will exist in the implementation. This idea is formalised by the following lemma.

Lemma 8.1. *Suppose Φ is output-distinguishable and Z and Z' are completely-defined. Let t be a test function of Z and $Y \subseteq \Phi^*$. If for all $s \in t(Y)$, $f_Z(s) = f_{Z'}(s)$ then $LR_Z \cap pref(Y) = LR_{Z'} \cap pref(Y)$.*

Proof. Let $p \in Y$ and $p_1 \in pref(p)$. Since Z and Z' are completely-defined, from $\mathrm{f}_Z(t(p)) = \mathrm{f}_{Z'}(t(p))$ it follows that $\mathrm{f}_Z(t(p_1)) = \mathrm{f}_{Z'}(t(p_1))$. Then, by induction on the length of p_1, it follows that $p_1 \in LR_Z$ if and only if $p_1 \in LR_{Z'}$. (A more detailed proof of this statement can be found in [29].) □

9. The test suite

Now, suppose that every sequence p in

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

is realisable. Then a test suite can be constructed by concatening each sequence in U with the set W_s, used to separate the states of Z, and then converting the resulting sequences of processing functions into input sequences. Thus the test suite produced will be $t(UW_s)$ for some test function t of Z. The following results validate this construction.

Lemma 9.1. *Let $x_1, x_2 \in LR_Z$, $q_1, q_2 \in Q$ such that $F^*(q_0, x_1) = q_1$ and $F^*(q_0, x_2) = q_2$. Suppose W_s separates between q_1 and q_2 in Z. If for all $s \in t(\{x_1, x_2\}W_s)$, $f_Z(s) = f_{Z'}(s)$ then there exist $q'_1, q'_2 \in Q'$ such that $F'^*(q'_0, x_1) = q'_1$, $F'^*(q'_0, x_2) = q'_2$ and W_s distinguishes between q'_1 and q'_2 in $A_{Z'}$.*

Proof. By Lemma 8.1, $x_1, x_2 \in LR_{Z'}$, so there exist such q'_1 and q'_2. Let $m_1, m_2 \in M$ be the memory values computed along x_1 and x_2 when Z receives the input sequences $t(x_1)$ and $t(x_2)$, respectively. Since W_s separates between q_1 and q_2 in Z, there exist $p_1 \in LR(q_1, m_1) \cap W_s$ and $p_2 \in LR(q_2, m_2) \cap W_s$ such that dom $p_1 \cap$ dom $p_2 \neq \emptyset$. Since for all $s \in t(\{x_1, x_2\}W_s)$, $f_Z(s) = f_{Z'}(s)$, by Lemma 8.1, $LR_Z(q_1, m_1) \cap W_s = LR_{Z'}(q'_1, m_1) \cap W_s$ and $LR_Z(q_2, m_2) \cap W_s = LR_{Z'}(q'_2, m_2) \cap W_s$. Thus $p_1 \in LR_{Z'}(q'_1, m_1) \cap W_s$ and $p_2 \in LR_{Z'}(q'_2, m_2) \cap W_s$. Then $p_1 \in L_{A_{Z'}}(q'_1) \cap W_s$ and $p_2 \in L_{A_{Z'}}(q'_2) \cap W_s$. Since Z' is deterministic, $p_1 \notin L_{A_{Z'}}(q'_2)$ and $p_2 \notin L_{A_{Z'}}(q'_1)$. Thus W_s distinguishes between q'_1 and q'_2 in $A_{Z'}$. □

Therefore, from the implementation passing the tests we can deduce that states that are separable in the specification are implemented as distinct states.

Lemma 9.2. *Let $q \in Q$ and $x \in V(q)$. Suppose $p_q x \in LR_Z$. If for all $s \in t(S_r W_s \cup \{p_q\} pref(x) W_s)$, $f_Z(s) = f_{Z'}(s)$ 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 Lemma 8.1, 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 9.1, since W_s pairwise separates 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. □

A direct consequence of the above lemma is that if *Fail* was reachable, it would be reached by some sequence from U.

Lemma 9.3. *Suppose the Fail state of $A_{P(Z,Z')}$ can be reached by some sequence from L_{A_Z}. If for all $s \in t(UW_s)$, $f_Z(s) = f_{Z'}(s)$, then Fail can be reached by some sequence from U.*

Proof. Suppose *Fail* is reached by $p \in L_{A_Z}$. Then there exist $p_1 \in L_{A_Z} \cap L_{A_{Z'}}$, $\phi \in \Phi$ and $(q_1, q'_1) \in Q \times Q'$ such that p_1 reaches (q_1, q'_1) in $A_{P(Z,Z')}$, $(q_1, \phi) \in \text{dom } F$ and $(q'_1, \phi) \notin \text{dom } F'$. Since $\epsilon \in S_r$, $p_1 \in S_r \Phi^*$. Let $i \geq 0$ be the minimum integer for which there exists a sequence in $S_r \Phi^i$ that reaches (q_1, q'_1) in $A_{P(Z,Z')}$. Let $p_2 = p_q x$ be such a sequence, $p_q \in S_r$, $x \in \Phi^i$. As $p_2 \in L_{A_Z}$, either p_2 is contained in $pref(V(q))$ or extends some sequence from $V(q)$, i.e. $x \in V(q)\Phi^*$. Since i is the minimum integer with the above property, the path in $A_{P(Z,Z')}$ formed by following x after p_q will be cycle-free and will not meet any state reached by sequences in S_r. Then, by Lemma 9.2, $x \in pref(V(q)) \setminus V(q)$. Thus, since $p_q x\phi \in L_{A_Z}$, $x\phi \in pref(V(q))$, so $p_2\phi \in U$. Since p_2 reaches (q_1, q'_1), $p_2\phi$ will reach *Fail* in $A_{P(Z,Z')}$. Thus the result follows. □

Lemma 9.4. *Suppose $U \subseteq LR_Z$. If for all $s \in t(UW_s)$, $f_Z(s) = f_{Z'}(s)$ then there is no path in LR_Z that reaches Fail in $A_{P(Z,Z')}$.*

Proof. We provide a proof by contradiction. Assume *Fail* can be reached by some sequence from LR_Z. By Lemma 9.3, *Fail* can be reached by some sequence from U. On the other hand, by Lemma 8.1, $LR_Z \cap U = LR_{Z'} \cap U$. Since $U \subseteq LR_Z$, it follows that $U \subseteq LR_{Z'}$. Since $U \subseteq L_{A_Z} \cap L_{A_{Z'}}$, no sequence in U can reach *Fail*. This provides a contradiction, as required. □

Consequently, the application of the sequences in $t(UW_s)$ to a faulty IUT will produce at least one output that does not conform to the specification. Thus $t(UW_s)$ is a valid test suite.

Theorem 9.1. *Suppose $U \subseteq LR_Z$. Then $f_Z = f_{Z'}$ if and only if for all $s \in t(UW_s)$, $f_Z(s) = f_{Z'}(s)$.*

Proof. Follows from Lemmas 6.1 and 9.4. □

When all the states of Z are r-reachable and pairwise separable, S_r is an r-state cover of Z, W_s is a separating set of Z and all states of Z are known to be pairwise separable by W_s, the test suite becomes

$$t(S_r \Phi[n' - n + 1]W_s),$$

This corresponds to the test suite produced in [26] for *controllable* DSXM specifications and the method reduces to an extension of the W-method [10] to DSXMs.

For $U = S_r \Phi[n' - n + 1]$, the number of sequences in UW_s is at most $n^2 \cdot k^{n'-n+1}$ and the total length of all sequences in UW_s is at most $n^2 \cdot n' \cdot k^{n'-n+1}$, where $k = card(\Phi)$ [10]. In the worst case, when $S_r = W_s = \{\epsilon\}$, 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 separable.

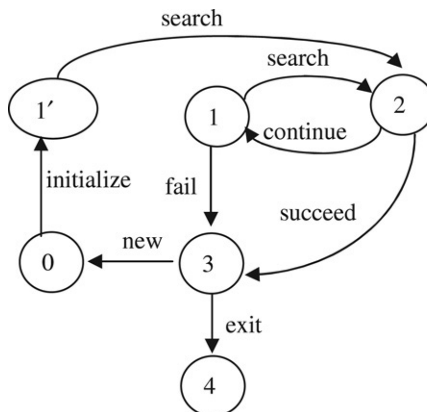

Fig. 3. The revised transition diagram.

10. The test generation method: Summary

We can now assemble the concepts and results given so far and provide a brief summary of the proposed method. The following pre-requisites need to be in place:

1. The specification Z is a DSXM. Normally, the associated FA of Z will be minimal, but this restriction is not imposed.
2. The type Φ of the specification is output-distinguishable. Unlike in the previous literature on stream X-machine based testing, the input-uniformity (or the stronger input-completeness) condition is not required.
3. The implementation under test can be modelled by a DSXM Z' with the same type Φ as the specification and the number of states in Z' is bounded by an estimated integer n', larger than or equal to the number n of states in Z. Since, unlike in [29], Z' is not required to be controllable, the size of the implementation model will be comparable to the size of the specification, i.e. $n' \approxeq n$.

Under these conditions, Theorem 9.1 guarantees that the test suite $t(UW_s)$ will determine the correctness of the implementation under test with regard to the specification Z.

11. Discussion

This paper provides a new variant of the stream X-machine based testing method that is no longer dependent on the size of a controllable DSXM model of the implementation. Instead, it requires that the specification is such that all the elements of the set $U = \bigcup_{q \in Q_r} \{p_q\} pref(V(q))$, constructed as in Section 7, are *realisable* sequences of processing functions. We discuss how this requirement can be satisfied in practical applications of the method.

Consider again our example and suppose that only non-empty strings of characters are allowed. Therefore, the system will have to perform at least one *search* before deciding whether the search has succeeded or failed. Thus the path *initialize fail*, which is contained in U, will not be realisable and, consequently, the requirement will not be met.

One possible way to address this problem would be to remove the non-realisable path(s) from the associated automaton and modify the state-transition diagram of the specification accordingly. The revised state-transition diagram for our example, obtained by eliminating the non-realisable path *initialize fail* from the language accepted by the automaton, is as represented in Fig. 3. Since the specification has been changed, the set U will have to be reconstructed from the new state-transition diagram. Consequently, this approach may give rise to an iterative process, in which, at each step, the non-realisable paths found at the previous step are removed from the state-transition diagram and U is reconstructed accordingly. However, for the applications we have looked at so far, only one iteration was sufficient. Finding design restrictions to ensure that this process will end after a finite number of iterations can be the subject of further investigations.

An alternative solution to this problem is based on the observation that a path may or may not be realisable, depending on the values of the data processed by the system. In the case of a stream X-machine specification, there are two kinds of data that affect the evolution of the system: the input symbols and the initial memory values. The input

symbols correspond to the values entered by the user, whereas the initial memory represents the data originally stored by the system; this may contain actual data, that may change in time, as well as system constants. The appropriate input symbols to drive a path in the specification are chosen through the construction of a test function (Definition 8.1). Furthermore, it can be observed that from the IUT passing the tests in the test suite we can deduce that the associated automaton of the specification and that of the IUT model are equivalent. Therefore, the correctness of the IUT does not depend on the choice of the initial memory. Consequently, the definition of a test function can be further generalized so that it associates a sequence of processing functions p with a pair consisting of the initial memory value used to test p and the sequence of inputs to drive p (or some prefix of p, as in Definition 8.1). In our example, the minimum or maximum allowed length of the string s will normally be a system constant whose value can be modified for testing purposes and changed back to the required value after testing has been completed. This approach does not involve redesigning the specification, but will add complexity to the generation process since (input sequence, initial memory) pairs, rather than mere input sequences, are searched.

A mixed approach, in which the first method (just the first iteration) is used to redesign the specification in order to make it testable and the remaining (if any) non-realisable paths are tested using the second method, may provide the best solution to the problem.

Finally, let us comment on the complexity of the test generation process. This involves converting each sequence of processing functions $p \in UW_s$ into a pair $t(p) = (m_p, s_p)$ consisting of an initial memory value m_p (as pointed out earlier, this may vary from one sequence to another) and a sequence of inputs s_p that, when applied in m_p, will force the machine to follow the path p. Both m_p and s_p will consist of data values processed by the system, so the amount of effort required for the generation of $t(p)$ will be, at worst, proportional to the size of the system data.[7] Thus, the complexity of the whole process will be, at worst, proportional to the number of sequences in UW_s and the size of the system data. However, various techniques could be used to reduce the effort involved in the generation process. A commonly used technique is symbolic execution [32,11]. Rather than running the program on actual input values, symbolic computation derives a set of constraints in terms of the input variables (in our case initial memory and input symbols) which describe the conditions necessary for the traversal of a given path. Constraint satisfaction problems are in general NP-complete [11]. However, linear programming techniques can be applied if the constraints are linear [11]. If this is not the case, metaheuristic search techniques can be used instead to attempt to find a solution. The application of search-based techniques in structural test data generation has been extensively investigated in recent years [34] and the proposed methods could be adapted for test generation from a graph-based model like the stream X-machine.

The method proposed here is an advance on previous stream X-machine based approaches. Consider, for comparison, the DSXM based test generation method devised in [29]. This requires a *controllable* model of the implementation, whose size may be, at worst, proportional to the size of the system data. Thus, in this case, both the test suite and the generation process will depend on the size of the system data. In contrast, the method proposed here will produce test suites whose size is typically unrelated to the system data. As the generation process only involves the specification but not the actual system and is normally automated, the effort spent in the application of the test data to the implementation and hence the size of the test suite is much more significant.

12. Conclusions

Stream X-machines (SXMs) are a form of extended finite state machine that can be naturally used for system specification. Furthermore, one of the great benefits of using stream X-machines to specify a system is that it is possible to produce a finite test suite that determines correctness as long as certain design for test conditions hold.

Originally, the work on testing from deterministic SXMs (DSXMs) has included the condition that the specification is controllable: all paths through the specification DSXM are feasible. This condition is quite restrictive and, in particular, is seldom met by data processing-oriented systems. Recent work [29] has replaced this restriction with a much laxer condition, called input-uniformity. However, the size of the test suite produced by this improved method strongly depends on the (estimated) upper bound on the number of states of a *controllable* model of the IUT. While this assumption is in general reasonable for most interactive systems, it may produce unmanageable test suites for even simple data processing-oriented applications.

[7] In reality, only the data that may affect the flow in the stream X-machine diagram need to be examined.

The new method presented in this paper is no longer dependent on the size of a controllable DSXM model of the implementation. It also removes the input-uniformity condition required by the previous approach. Consequently, in data processing-oriented applications, the new method can drastically reduce the size of the test suite generated at the expense of a more complex process of translating sequences of processing functions into sequences of inputs. In data processing-oriented applications this trade-off is usually more than justified. Furthermore, alternative techniques, such as program slicing [1], may be used to reduce the complexity of this process. In comparison to [29], a slightly stronger distinguishability condition (called separability) between the states of the specification is used.

The method presented in this paper can also be used in structural (program based) testing [37] to achieve path coverage. In general, for even simple programs, path coverage (in the strictest definition of the term) cannot be achieved since the number of paths is, at best, very large and, at worst, infinite. Traditionally, a way around this is to choose equivalence classes [35] of paths. For example, two paths can be considered equivalent if they differ only in the number of loop traversals [37]. This will lead to two classes, one for 0 loop traversals and one for $k > 0$ traversals. A more refined partitioning based on the same criterion may also be chosen. However, a basic problem with this rather static way of partitioning paths into classes, is that the criterion (number of loops) does not take into account the processing of the program data (memory variables and data structures). On the contrary, our approach is based on both the control structure of the program and its data processing. Further work may also involve extending the method to allow testing of the processing functions to be integrated into the testing of the overall system, following the approach used in [22] for controllable specifications. The extension of the method to *non-deterministic* stream X-machine specifications [28,16,17] 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] D. Binkley, K.B. Gallagher, Program slicing, Advances in Computers 43 (1996) 1–50.
[2] J. Aguado, T. Bălănescu, T. Cowling, M. Gheorghe, M. Holcombe, F. Ipate, P systems with replicated rewriting and stream X-machines (Eilenberg machines), Fundamenta Informaticae 49 (1–3) (2002) 17–33.
[3] T. Bălănescu, Generalized stream X machines with output delimited type, Formal Aspects of Computing 12 (2000) 473–484.
[4] T. Bălănescu, T. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, C. Vertan, Communicating stream X-machines are no more than X-machines, Journal of Universal Computer Science 5 (1999) 494–507.
[5] T. Bălănescu, M. Gheorghe, M. Holcombe, F. Ipate, Testing collaborative agents defined as stream X-machines, in: Advances in Artificial Life, Proceedings of the 6th European Conference ECAL, 10–14 September, Prague, Czech Republic, Springer, Berlin, 2001, pp. 296–305.
[6] T. Bălănescu, M. Gheorghe, M. Holcombe, F. Ipate, Eilenberg P Systems, in: Gh. Paun, G. Rozenberg, A. Salomaa, C. Zandron (Eds.), Membrane Computing. International Workshop, WMC-CdeA 2002, August 2002, Curtea de Arges Romania, in: LNCS, vol. 2597, Springer-Verlag, Berlin, 2003, pp. 43–57.
[7] J. Barnard, J. Whitworth, M. Woodward, Communicating X-machines, Information and Software Technology 38 (1996) 401–407.
[8] F. Bernardini, M. Gheorghe, M. Holcombe, P X systems = P systems + X machines, Natural Computing 2 (3) (2003) 201–213.
[9] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed, S. Vanak, Testing methods for X-machines, a review, Formal Aspects of Computing 18 (1) (2006) 3–30.
[10] T.S. Chow, Testing software design modelled by finite state machines, IEEE Transactions on Software Engineering 4 (3) (1978) 178–187.
[11] L. Clarke, A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering 2 (3) (1976) 215–222.
[12] A. Cowling, H. Georgescu, C. Vertan, A structured way to use channels for communication in X-machine systems, Formal Aspects of Computing 12 (6) (2000) 458–500.
[13] S. Eilenberg, Automata, Languages and Machines, vol. A., Academic Press, New York, 1974.
[14] 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.
[15] H. Georgescu, C. Vertan, A new approach to communicating X-machines, Journal of Universal Computer Science 6 (5) (2000) 490–502.
[16] R.M. Hierons, M. Harman, Testing conformance to a quasi-non-deterministic stream X-machine, Formal Aspects of Computing 12 (6) (2000) 423–442.
[17] R.M. Hierons, M. Harman, Testing conformance of a deterministic implementation to a non-deterministic stream X-machine, Theoretical Computer Science 323 (1–3) (2004) 191–233.
[18] M. Holcombe, X-machines as a basis for dynamic system specification, Software Engineering Journal 3 (1988) 69–76.
[19] M. Holcombe, F. Ipate, Correct Systems: Building a Business Process Solution, Springer Verlag, Berlin, 1998.

[20] M. Holcombe, F. Ipate, A. Grondoudis, Complete Functional Testing of Safety-Critical Systems, in: Proceedings of the 2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies, 1–3 November, Daytona Beach, FL, USA, Elsevier, Oxford, 1995, pp. 199–204.
[21] F. Ipate, On the minimality of stream X-machines, The Computer Journal 46 (3) (2003) 295–306.
[22] F. Ipate, Complete deterministic stream X-machine testing, Formal Aspects of Computing 16 (4) (2004) 374–386.
[23] F. Ipate, M. Holcombe, A method for refining and testing generalized machine specifications, International Journal of Computer Mathematics 68 (1998) 197–219.
[24] F. Ipate, M. Holcombe, An integrated refinement and testing method for stream X-Machines, Applicable Algebra in Engineering, Communication and Computing 13 (2) (2002) 67–91.
[25] F. Ipate, M. Holcombe, Testing conditions for communicating stream X-machine systems, Formal Aspects of Computing 13 (6) (2002) 431–446.
[26] F. Ipate, M. Holcombe, An integration testing method that is proved to find all faults, International Journal of Computer Mathematics 63 (1997) 159–178.
[27] 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.
[28] F. Ipate, M. Holcombe, Generating test sequences from non-deterministic generalized stream x-machines, Formal Aspects of Computing 12 (6) (2000) 443–458.
[29] F. Ipate, Testing against a non-controllable stream x-machine using state counting, Theoretical Computer Science 353 (1–3) (2006) 291–316.
[30] 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.
[31] 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.
[32] J. King, Symbolic execution and program testing, Communications of the ACM 19((7) (1976) 385–394.
[33] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines — a survey, Proceedings of the IEEE 84 (8) (1996) 1090–1123.
[34] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.
[35] T.J. Ostrand, M.J. Balcer, The category-partition method for specifying and generating functional tests, Communication of the ACM 31 (6) (1989) 667–686.
[36] A. Petrenko, N. Yevtushenko, G.v. Bochmann, Testing deterministic implementations from nondeterministic FSM specifications, in: Proc. of 9th International Workshop on Testing of Communicating Systems, IWTCS'96, 1996, pp. 125–140.
[37] M. Roper, Software Testing, McGraw-Hill, 1994.