

Complete deterministic stream X-machine testing

Florentin Ipate

Department of Computer Science and Mathematics, University of Pitesti, Romania

Abstract. One of the strengths of using *stream X-machines* to specify a system is that, under certain well defined conditions, it is possible to produce a test set that is guaranteed to determine the correctness of an implementation. However, the existing method assumes that the implementation of each processing function is proved to be correct before the actual testing can take place, so it only test the system *integration*. This paper presents a new method for generating test sets from a deterministic stream X-machine specification that generalises the existing integration testing method. This method no longer requires the implementations of the processing functions to be proved correct prior to the actual testing. Instead, the testing of the processing functions is performed along with the integration testing.

Keywords: Testing; Test set generation; Formal specification; Stream X-machines; Finite state machines

1. Introduction

One approach to formally specifying a system is to use a form of extended finite state machine called a *stream X-machine* [HoI98, IpH96]. A stream X-machine (*SXM* for short) is a type of X-machine [Eil74, HoI88, HoI98] that describes a system as a finite set of states, each with 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*). Thus, stream X-machines can combine the dynamic features of finite state machines with data structures, thus sharing the benefits of both these worlds. Various case studies [HoI98, FHI95, KEK00] 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 [KeK00]. The *refinement* of stream X-machines [IpH98a, IpH02a] as well as various subclasses of stream X-machines [IpH96, BGH00, BGH01, Ghe00] have also been investigated. The minimality issue has also been investigated in the context of stream X-machines [Ipa03]. Furthermore, several models of *communicating stream X-machines* have been devised and their applicability to real applications has been demonstrated [BWW96, BCG09, CHV00, GeV00, IpH02b]. Communicating stream X-machines have also been used for modelling P-systems [ABC02].

One of the strengths of using stream X-machines to specify a system is that, under certain well defined conditions, it is possible to produce a test set that is guaranteed to determine the correctness of an implementation [IpH97, HoI98, MIG95]. These conditions fall into two categories. Firstly, the stream X-machine specification has

to meet some “design for test conditions” viz. input-completeness and output-distinguishability [IpH97, HoI98]. Secondly, the method assumes that the processing functions are correctly implemented and reduces the testing of a stream X-machine to the testing of its associated finite automaton. Therefore, it is fair to say that the method only tests the *integration* of the processing functions implementations into the system implementation. In practice, the correctness of the implementation of a processing function is checked by a separate process [HoI98, IpH98b]: depending on the nature of the function, it can be tested using the same method or alternative functional methods, for example category partition testing [OsB89] or a variant.

The method (called in what follows *SXM integration testing*) was first developed in the context of deterministic stream X-machines (DSXM integration testing) [IpH97, HoI98] and then extended to the non-deterministic case (NSXM integration testing) [IpH00]. Alternative design for test conditions have also been investigated [Bal01]. The method in which, initially, only equivalence testing was considered, has also been extended to address conformance testing [HiH00].

Unlike the traditional extended finite state machine testing approaches [ChK93, LeY96, WaL93], the SXM integration testing method does not involve the construction of the equivalent finite state machine (whose states are the state/memory pairs of the original stream X-machine), and therefore does not rely on the finiteness of the memory and avoids the state explosion problem associated with this construction. Instead, however, it assumes that the implementation of each processing function can be tested *in isolation* from the rest of the system. This is not always a realistic assumption since the implementations of the processing functions are not always separate units of code (functions, procedures, etc.) that can be separated from the rest of the implementation.

This paper presents a new method for generating test sets from a deterministic stream X-machine specification that generalises the existing DSXM integration testing method. The new method (called *complete DSXM testing*) no longer requires the implementations of the processing functions to be proved correct before the actual testing can take place. Instead, the testing of the processing functions is performed along with the integration testing. Therefore, the test set generated by this method will be made up of two components: one that tests the processing functions and the other that tests their integration. This latter component is in fact the test set generated by the DSXM integration testing method.

The paper is structured as follows. Sections 2 and 3 introduce basic concepts of finite automata and stream X-machines, respectively. The (existing) DSXM integration testing method is presented in Section 4. The theoretical basis for the (new) complete DSXM testing method is given in section 5; its applicability is discussed in Section 6. Conclusions and directions for future work are given in Section 7.

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 $a, b \in A^*$, ab denotes the concatenation of sequences a and b . a^n is defined by $a^0 = \epsilon$ and $a^n = a^{n-1}a$ for $n \geq 1$. For $U, V \subseteq A^*$, $UV = \{ab \mid a \in U, b \in V\}$; U^n is defined by $U^0 = \{\epsilon\}$ and $U^n = U^{n-1}U$ for $n \geq 1$.

For a sequence $a \in A^*$, $length(a)$ denotes the number of elements of a (in particular $length(\epsilon) = 0$).

For a (partial) function $f : A \rightarrow B$, $dom(f)$ denotes the domain of f , i.e. the subset of A for which f is defined. $Im(f)$ denotes the image of f , i.e. $Im(f) = \{f(a) \mid a \in A\}$. For $U \subseteq A$, $f(U) = \{f(a) \mid a \in U\}$. For $U \subseteq A$, $f \upharpoonright U$ denotes the restriction of f to U , i.e. $f \upharpoonright U : U \rightarrow B$ is defined by $f \upharpoonright U(a) = f(a)$, $\forall a \in U$.

For two (partial) functions $f, g : A \rightarrow B$, we use $f \subseteq g$ to denote that $f(a) = g(a)$, $\forall a \in dom(f)$.

For n sets A_1, \dots, A_n , $\pi_i : A_1 \times \dots \times A_n \rightarrow A_i$ denotes the projection function, for $1 \leq i \leq n$.

For a finite set A , $card(A)$ denotes the number of elements in A .

2. Finite automata

This section defines the finite automaton and related concepts and results to be used later in the paper.

Definition 2.1. A *finite automaton* (FA for short) A is a tuple (Σ, Q, F, I, T) , where:

- Σ is the finite *input alphabet*;
- Q is the finite *set of states*;
- F is the (partial) *next state function*, $F : Q \times \Sigma \rightarrow 2^Q$;
- I and T are the sets of *initial* and *terminal states* respectively, $I \subseteq Q$, $T \subseteq Q$.

F is usually described by a transition diagram. If $q, q' \in Q$, $\sigma \in \Sigma$ and $q' \in F(q, \sigma)$ we say that σ is an *arc* from q to q' and write $\sigma : q \rightarrow q'$.

Definition 2.2. An FA is called *deterministic* if:

- There is one initial state, i.e.

$$I = \{q_0\};$$

- F maps each state/input pair into at most one state, i.e.

$$F : Q \times \Sigma \longrightarrow Q.$$

In what follows we will only refer to deterministic FAs with all states terminal ($T = Q$), denoted by a tuple (Σ, Q, F, q_0) .

Definition 2.3. The next state function can be extended to a (partial) function $F^* : Q \times \Sigma^* \longrightarrow Q$ defined by:

- $F^*(q, \epsilon) = q, \forall q \in Q$;
- $F^*(q, s\sigma) = F(F^*(q, s), \sigma), \forall q \in Q, s \in \Sigma^*, \sigma \in \Sigma$.

Definition 2.4. For $q \in Q$, the *language accepted by A in q* , denoted by $L_A(q)$, is defined by:

$$L_A(q) = \{s \in \Sigma^* \mid (q, s) \in \text{dom}(F^*)\}.$$

The language accepted by A in q_0 is simply called the *language accepted by A* and is denoted by L_A .

Definition 2.5. A state $q \in Q$ is called *accessible* if $\exists s \in \Sigma^*$ with $F^*(q_0, s) = q$. A is called *accessible* if $\forall q \in Q$, q is accessible.

Definition 2.6. For $U \subseteq \Sigma^*$, two states $q_1, q_2 \in Q$ are called *U -equivalent* if $L_A(q_1) \cap U = L_A(q_2) \cap U$. Otherwise q_1 and q_2 are called *U -distinguishable*. If $U = \Sigma^*$ then q_1 and q_2 are simply called *equivalent* or *distinguishable*. A is called *reduced* if $\forall q_1, q_2 \in Q, ((q_1 \neq q_2) \implies (q_1 \text{ and } q_2 \text{ are distinguishable}))$.

Definition 2.7. A deterministic FA, A , is called *minimal* if any other FA that accepts the same language as A has at least the same number of states as A .

Theorem 2.1. A is minimal if and only if A is accessible and reduced.

This is a well known result; for a proof see for example [Eil74].

Definition 2.8. Let $A = (\Sigma, Q, F, q_0)$ and $A' = (\Sigma, Q', F', q'_0)$ be two deterministic FAs having the same input alphabet. Then a bijective function $g : Q \longrightarrow Q'$ is called an *isomorphism* if:

- $g(q_0) = q'_0$;
- $g(F(q, \sigma)) = F'(g(q), \sigma), \forall q \in Q, \sigma \in \Sigma$.

That is, an isomorphism is a function that renames the states of a FA.

Theorem 2.2. For two minimal deterministic FAs A and A' , $L_A = L_{A'}$ if and only if A and A' are isomorphic.

This is a well known result; for a proof see, for example, [Eil74]. Techniques for constructing the minimal FA that accepts a given language also exist; for more detail see, for example, [Eil74].

We now turn our attention to FA testing and, in particular, to the generation of test sequences from a FA specification.

Given a FA specification A and a class of implementations C , a test set is a set of input sequences that, when applied to any implementation A' in the class C , will detect any response in A' that does not conform to the response specified by A .

Definition 2.9. Let A be a deterministic FA and C a set of deterministic FAs having the same input alphabet Σ as A . Then a finite set $Y \subseteq \Sigma^*$ is called a *test set of A w.r.t. C* if $\forall A' \in C, (L_A \cap Y = L_{A'} \cap Y \implies L_A = L_{A'})$.

The class C is identified by the assumptions we can make about the implementation A' . In principle, no information is available about the implementation, but in this case a test set may not exist for even very simple FA specifications.

There are a number of more or less realistic assumptions that one can make about the form and size of the implementation and these, in turn, give rise to different techniques for generating test sets [LeY96]. One of the least restrictive assumptions refers to the number of states of A' and is the basis for the *W-method* [Cho78, BGI03]: the difference between the number of states of the implementation and that of the specification has to be at most

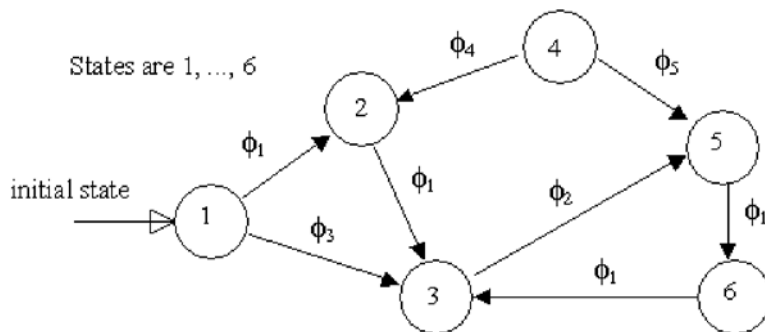


Fig. 1. The state transition diagram of an X-machine

k , a positive integer estimated by the tester. The W -method was first presented by Chow [Cho78] in the context of input/output finite state machines (i.e. for which F is of the form $F : Q \times \Sigma \rightarrow 2^{Q \times \Gamma}$, where Γ is the output alphabet) that are *completely specified* (i.e. $\emptyset \notin Im(F)$) and was later extended to *partially specified* input/output finite state machines and finite automata [BGI03]. This latter form of the W method is presented here, the context being that of finite automata.

The following concepts are from [BGI03]:

Definition 2.10. $S \subseteq \Sigma^*$ is called a *state cover* of A if $\epsilon \in S$ and $\forall q \in Q \setminus \{q_0\}, \exists s \in S$ such that $F^*(q_0, s) = q$.

Definition 2.11. $P \subseteq \Sigma^*$ is called a *transition cover* of A if $S \cup S\Phi \subseteq P$ for some state cover S of A .

Definition 2.12. $W \subseteq \Sigma^*$ is called a *characterisation set* of A if any two distinct states of A , $q_1, q_2 \in Q$, $q_1 \neq q_2$, are W -distinguishable.

Note that a state cover, a transition cover and a characterisation set exist if A is minimal. These concepts are illustrated in Example 4.2.

The following result is the theoretical basis for the W -method in the context of deterministic finite automata:

Theorem 2.3. [BGI03] Let A be a deterministic FA having input alphabet Σ , n the number of states of A , $m \geq n$ and C_m the set of deterministic FAs having input alphabet Σ whose number of states does not exceed m . If P is a transition cover and W a characterisation set of A then $Y_{m-n} = P(\Sigma^{m-n} \cup \dots \cup \{\epsilon\})(W \cup \{\epsilon\})$ is a test set of A w.r.t. C_m .

3. Stream X-machines

In this section the stream X-machine and other basic concepts related to it are defined.

In its essence an *X-machine* is like a finite state machine but with one important difference. Instead of using abstract symbols, the labels of the transitions are (*partial*) *functions* that operate on a *basic data set* X . The set of these (*partial*) functions, Φ , is called the *type* of the machine and represents the elementary operations that the machine is capable of performing.

The computation of the machine starts in a given initial state (control state) and a given state of the system's underlying data type X (the data state). In Fig. 1, for example, there are a number of paths that can be traced out from the initial state and each edge is labelled by a function: ϕ_1, ϕ_2 , etc. Sequences of functions are thus derived from each path in the state space and these may be composed to produce a function that may be defined on the data state. This is then applied to the value x providing that the composed function is defined on x . This then gives a new value, $x \in X$ for the data state and a new control state. Usually, the machine is deterministic so that at any moment there is only one possible function defined (that is the domains of the functions that emerge from any state are mutually disjoint).

Those X-machines in which all data are triples consisting of a stream of input symbols, a stream of output symbols and an internal memory value are called *stream X-machines* and are defined formally next. The basic idea is that the machine has some internal memory, M , and the stream of inputs determine, depending on the current state of control and the current state of the memory, the next control state, the next memory state and the output value.

Definition 3.1. A *stream X-Machine* (SXM for short) is a tuple

$$Z = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0),$$

where:

- Σ and Γ are finite sets called the *input alphabet* and *output alphabet* respectively;
- Q is the finite set of *states*;
- M is a (possibly) infinite set called *memory*;
- Φ is the *type* of Z , a finite set of distinct *processing functions* that the machine can use; a processing function is a non-empty (partial) function of the form

$$\phi : M \times \Sigma \longrightarrow \Gamma \times M;$$

- F is the (partial) *next state function*,

$$F : Q \times \Phi \longrightarrow 2^Q;$$

As for finite automata, F is usually described by a *state-transition diagram*.

- I and T are the sets of initial and terminal states respectively,

$$I \subseteq Q, T \subseteq Q;$$

- m_0 is the initial memory value,

$$m_0 \in M.$$

Thus, SXMs are X-machines for which the processing functions have the form $\phi : M \times \Sigma \longrightarrow \Gamma \times M$, i.e. each such function will read an input symbol, discard it and produce an output symbol while (possibly) changing the value of the memory.

It is sometimes helpful to think of an X-machine as a finite automaton with the arcs labelled by functions from the type Φ . The automaton $A_Z = (\Phi, Q, F, I, T)$ over the alphabet Φ is called *the associated FA* of Z . Analogously to finite automata, if for $q, q' \in Q, \phi \in \Phi, F(q, \phi) = q'$, then ϕ is called an *arc* of Z from q to q' , denoted $\phi : q \rightarrow q'$.

A completely specified SXM is one in which there is at least one possible transition for any triplet $q \in Q, m \in M, \sigma \in \Sigma$. This is now defined.

Definition 3.2. A SXM $Z = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ is called *completely specified* if $\forall q \in Q, m \in M, \sigma \in \Sigma, \exists \phi \in \Phi$ such that $((m, \sigma) \in \text{dom}(\phi) \text{ and } (q, \phi) \in \text{dom}(F))$.

Definition 3.3. Given a sequence $p \in \Phi^*$, p induces the (partial) function

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

defined as follows:

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

Thus $|p|$ shows the correspondence between a (memory, input string) pair and the (output string, memory) pair produced by the application, in turn, of the processing functions in the sequence p .

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 3.4. An SXM Z is called *deterministic* if the following hold.

- The associated FA of the machine is deterministic, i.e.
 - Z has only one initial state, i.e.

$$I = \{q_0\};$$

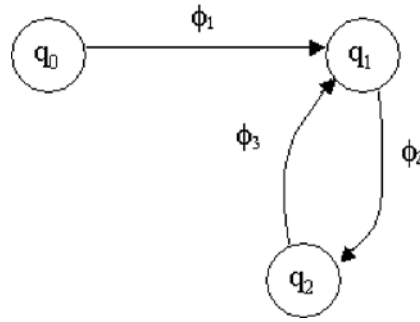


Fig. 2. The associated FA of Z

– The next state function of Z maps each pair (state, processing function) onto at most one state, i.e.

$$F : Q \times \Phi \longrightarrow Q;$$

- Any two distinct processing functions that label arcs emerging from the same state have disjoint domains, i.e. $\forall \phi_1, \phi_2 \in \Phi, ((\exists q \in Q \text{ with } (q, \phi_1), (q, \phi_2) \in \text{dom}(F)) \implies (\phi_1 = \phi_2 \text{ or } \text{dom}(\phi_1) \cap \text{dom}(\phi_2) = \emptyset))$.

Note that if a deterministic SXM is completely specified then there is exactly one transition for any triplet $q \in Q, m \in M, \sigma \in \Sigma$.

Since stream X-machines are used in this paper as a basis for testing, it is normal to assume that every state is terminal, i.e. $T = Q$. This basically means that the output produced by the machine can be viewed in any of its states, even though the machine is only allowed to terminate its computation in certain states.

Thus, in what follows, we will refer to *deterministic SXMs* (*DFSXMs* for short) with all states terminal, denoted by a tuple $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$. The associated FA is then a tuple $A_Z = (\Phi, Q, F, q_0)$.

Example 3.1. A DSXM $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ with $\Sigma = \{a, b\}$, $\Gamma = \{x, y, z, w\}$, $Q = \{q_0, q_1, q_2\}$, $M = \{0, 1\}$, $m_0 = 0$, $\Phi = \{\phi_1, \phi_2, \phi_3\}$, F as represented in Fig. 2 and $\phi_1, \phi_2, \phi_3 : M \times \Sigma \longrightarrow \Gamma \times M$ defined by:

$$\phi_1(m, a) = (x, m), m \in M;$$

$$\phi_2(0, a) = (y, 0),$$

$$\phi_2(1, a) = (z, 1);$$

$$\phi_3(m, b) = (w, 1 - m), m \in M$$

will be used in illustrations later in the paper.

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 *function computed* by the machine, as defined next.

Definition 3.5. Given a DSXM Z , the (partial) function $f_Z : \Sigma^* \longrightarrow \Gamma^*$ defined by:

$$f_Z(s) = g \text{ if } \exists p \in \Phi^*, m \in M \text{ such that } (q_0, p) \in \text{dom}(F^*) \text{ and } | p | (m_0, s) = (g, m)$$

is called the *function computed* by Z . We say that Z *computes* f_Z .

Note that f_Z is a function since Z is deterministic. However, in general, a (non-deterministic) SXM may compute a relation rather than a function since the application of an input sequence may produce more than one output sequence. This paper will, however, deal only with the deterministic case.

If a DSXM Z is completely specified then f_Z is a total function.

4. DSXM integration testing

This section presents the theoretical basis for the DSXM integration testing method [IpH97, HoI98]. This generates a test set from a DSXM specification, providing that the system components (i.e. the processing functions)

are implemented correctly. Therefore, it is assumed that the implementation is a DSXM having the same type (processing functions) as the specification.

A *test set* is a finite set of input sequences constructed from the DSXM specification that produces identical results when applied to the specification and the implementation only if the specification and the implementation compute identical functions.

Definition 4.1. Let Z be a DSXM and C a set of DSXMs having the same input alphabet (Σ) and output alphabet (Γ) as Z . Then a finite set $X \subseteq \Sigma^*$ is called a *test set* of Z w.r.t. C if $\forall Z' \in C, (f_Z \mid X = f_{Z'} \mid X \implies f_Z = f_{Z'})$.

It is natural to assume that the implementation is a DSXM having the same input alphabet, output alphabet, memory and initial memory as the specification.

Definition 4.2. Two DSXMs Z and Z' are called *weak testing compatible* if they have identical input alphabets, output alphabets, memory sets and initial memory values.

Furthermore, as discussed in the introduction of the section, the implementation will be assumed to have the same type as the specification.

Definition 4.3. Two weak testing compatible DSXMs are called *testing compatible* if they have identical types.

The method also requires that the DSXM specification satisfies two conditions: input-completeness and output-distinguishability.

Definition 4.4. Φ is called *output-distinguishable* if $\forall \phi_1, \phi_2 \in \Phi, ((\exists m \in M, \sigma \in \Sigma \text{ with } \pi_1(\phi_1(m, \sigma)) = \pi_1(\phi_2(m, \sigma))) \implies \phi_1 = \phi_2)$.

This says that we must be able to distinguish between any two different processing functions by examining outputs. If we cannot then we will not always be able to tell them apart.

Definition 4.5. Φ is called *input-complete* if $\forall \phi \in \Phi, m \in M, \exists \sigma \in \Sigma$ such that $(m, \sigma) \in \text{dom}(\phi)$.

This condition ensures that any processing function can be exercised from any memory value using appropriate input symbols.

For Example 3.1., Φ is both input-complete and output-distinguishable.

These two conditions (output-distinguishability and input-completeness) are generally known as “design for test conditions” [HoI98, IpH97]. Without them, it would be extremely difficult to test a system properly. The output-distinguishability condition ensures that any processing function can be identified from the machine computation by examining the outputs produced. The input-completeness condition ensures that all sequences of processing functions in the associated FA can be exercised using appropriate inputs, so they can be tested against the implementation.

The basic idea of the method is to translate test sets of the associated FA into test sets of the DSXM specification. In order to do this, we need a mechanism, called a test function, that translates sequences of processing functions into sequences of inputs.

Definition 4.6. Let $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ be a DSXM having type Φ input-complete. Then a function $t : \Phi^* \longrightarrow \Sigma^*$ is called a *test function* of Z if the following hold:

- $t(\epsilon) = \epsilon$;
- for $\phi_1, \dots, \phi_n \in \Phi$, with $n > 0$, $t(\phi_1 \dots \phi_n) = \sigma_1 \dots \sigma_k$, where $\sigma_1, \dots, \sigma_k \in \Sigma$ are such that $(m_0, \sigma_1 \dots \sigma_k) \in \text{dom}(|\phi_1 \dots \phi_k|)$ and k is as follows:
 - $\phi_1 \dots \phi_n \in L_{A_Z} \implies k = n$;
 - $\phi_1 \dots \phi_n \notin L_{A_Z} \implies k = i + 1$, where $0 \leq i < n$ is such that $(\phi_1 \dots \phi_i \in L_{A_Z}$ and $\phi_1 \dots \phi_{i+1} \notin L_{A_Z})$.

In other words, for any sequence $v = \phi_1 \dots \phi_n$ of processing functions, $t(v)$ is a sequence of inputs that exercises the longest prefix $\phi_1 \dots \phi_i$ of v that is a path in the machine and, if $i < n$, also exercises ϕ_{i+1} , the function that follows after this prefix.

Note that since Φ is input-complete there always exist $\sigma_1, \dots, \sigma_k$ as above. Also note that, in general, a test function of Z is not uniquely determined, many different test functions may exist.

Example 4.1. The following values illustrate the construction of a test function for Z as in Example 3.1.:

$$t(\phi_1) = a,$$

$$\begin{aligned}
t(\phi_1\phi_2) &= aa, \\
t(\phi_1\phi_2\phi_3) &= aab, \\
t(\phi_1\phi_2\phi_3\phi_1) &= t(\phi_1\phi_2\phi_3\phi_1v) = aaba, \forall v \in \Phi^*.
\end{aligned}$$

The result below is the theoretical basis for DSXM integration testing.

Theorem 4.1. [IpH97, HoI98] Let Z be a DSXM having type Φ input-complete and output-distinguishable and C a set of DSXMs testing compatible with Z . If t is a test function of Z and $Y \subseteq \Phi^*$ a test set of A_Z w.r.t. A_C , where $A_C = \{A_{Z'} \mid Z' \in C\}$, then $X = t(Y)$ is a test set of Z w.r.t. C .

Since C is a set of DSXMs testing compatible with Z , it is assumed that the processing functions are implemented correctly (i.e. the implementation uses the same set of processing function as the implementation.) Therefore, the method only tests the *integration* of the processing functions into the whole system. The correctness of the implementations of the processing functions is checked by separate testing processes, as discussed in [IpH97, HoI98].

We can now use Theorem 4.1. and Theorem 2.3. to generate a test set of Z w.r.t. C_m , the set of DSXMs testing compatible with Z whose number of states does not exceed m . This is $X_{m-n} = t(Y_{m-n})$, where $Y_{m-n} = P(\Phi^{m-n} \cup \dots \cup \{\epsilon\})(W \cup \{\epsilon\})$, n is the number of states in Z , P is a transition cover and W a characterisation set of A_Z and t is a test function of Z .

The following example illustrates the construction of X_{m-n} for Z as in Example 3.1. and $m - n = 1$.

Example 4.2. $S = \{\epsilon, \phi_1, \phi_1\phi_2\}$,
 $P = S \cup S\Phi = \{\epsilon, \phi_1, \phi_2, \phi_3, \phi_1\phi_1, \phi_1\phi_2, \phi_1\phi_3, \phi_1\phi_2\phi_1, \phi_1\phi_2\phi_2, \phi_1\phi_2\phi_3\}$,
 $W = \{\phi_1, \phi_2\}$,
 $Y_1 = P(\Phi \cup \{\epsilon\})(W \cup \{\epsilon\})$,
 $X_1 = t(Y_1)$.

More detail about the applicability of the DSXM integration testing method may be found in [HoI98, IpH97].

5. Theoretical basis for complete DSXM testing

This section presents the theoretical basis for the complete DSXM testing method. Unlike integration testing, no assumption is made here regarding the correctness of the implementation of the processing functions. Therefore, the more general case, in which the specification and the implementation may have different types (are *weak* testing compatible), is considered.

When the functionality of a system is not specified for one or more inputs, it usually means that the implementation of the “missing” functionality is not important in the context of the specified system. Therefore, in order to establish that a DSXM Z' is an acceptable implementation of a (partially specified) DSXM specification Z , it is enough to show that $f_Z \subseteq f_{Z'}$.

The definition of a test set is revised to reflect this, more general, situation.

Definition 5.1. Let Z be a DSXM and C a set of DSXMs having the same input alphabet (Σ) and output alphabet (Γ) as Z . Then a finite set $X \subseteq \Sigma^*$ is called a *weak test set* of Z w.r.t. C if $\forall Z' \in C, (f_Z \mid X = f_{Z'} \mid X \implies f_Z \subseteq f_{Z'})$.

Unlike the definition of a test set, $f_Z \mid X = f_{Z'} \mid X$ above is a sufficient, but not a necessary condition. When $X \subseteq \text{dom}(f_Z)$ (in particular when Z is completely defined), the condition is also necessary. Furthermore, if Z is completely defined then X is a weak test set of Z w.r.t. C if and only if X is a test set of Z w.r.t. C . Conversely, if Z is completely specified and X is a weak test set of Z w.r.t. C then X is also a test set of Z w.r.t. C .

The output-distinguishability condition is also updated for the situation where the specification and the implementation may have different types.

Definition 5.2. Let $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $Z' = (\Sigma, \Gamma, Q', M, \Phi', F', q'_0, m_0)$ be two weak testing compatible DSXMs having types Φ and Φ' , respectively. Then Φ is called *output-distinguishable w.r.t. Φ'* if there exists an injective function $c : \Phi \rightarrow \Phi'$ such that the following holds: $\forall \phi \in \Phi, \phi' \in \Phi', ((\exists m \in M, \sigma \in \Sigma \text{ such that } \pi_1(\phi(m, \sigma)) = \pi_1(\phi'(m, \sigma))) \implies \phi' = c(\phi))$.

This says that, for a processing function ϕ in Φ , we must be able to identify a corresponding function $\phi' = c(\phi)$ in Φ' by examining outputs. Note that if $\Phi = \Phi'$ then c is the identity function and that Φ is output-distinguishable w.r.t. itself iff Φ is output-distinguishable.

In the conditions of Definition 5.2., we use $A_{Z'}^c$ to denote the FA (Φ', Q', F'_c, q'_0) , where F'_c is the restriction of F' to $Q' \times \text{Im}(c)$, i.e. $F'_c = F' \upharpoonright (Q' \times \text{Im}(c))$. Clearly, $L_{A_{Z'}^c} = L_{A_{Z'}} \cap \text{Im}(c)^*$, so $L_{A_{Z'}^c} \subseteq L_{A_{Z'}}$.

We also denote by $A_{Z'}^{-c} = (\Phi, Q', F'_{-c}, q'_0)$, the FA obtained by substituting each arc $c(\phi)$ in $A_{Z'}^c$ with ϕ , i.e. $F'_{-c}(q', \phi) = F'_c(q', c(\phi))$. Obviously, for $\phi_1, \dots, \phi_n \in \Phi$, $\phi_1 \dots \phi_n \in L_{A_{Z'}^{-c}}$ iff $c(\phi_1) \dots c(\phi_n) \in L_{A_{Z'}^c}$.

Since the method does not assume that the processing functions are correctly implemented, it will have to test their implementation in addition to their integration. Therefore, the test set generated by the method will be made up of two components: the *integration* test set (given in the previous section) and a set for testing the processing functions. This latter component will be called a *function test set*. The concepts necessary for its constructions are presented below.

Definition 5.3. Let $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $Z' = (\Sigma, \Gamma, Q', M, \Phi', F', q'_0, m_0)$ be two weak testing compatible DSXMs. Then for $\phi \in \Phi$ and $m \in M$, a finite set $\Sigma_m^\phi \subseteq \Sigma$ is called a *weak m test set of ϕ w.r.t. Φ'* if the following holds: $\forall \phi' \in \Phi', ((\forall \sigma \in \Sigma_m^\phi, (\pi_1(\phi(m, \sigma)) = \pi_1(\phi'(m, \sigma)))) \implies \phi \subseteq \phi')$.

That is, a weak m test set of ϕ w.r.t. Φ' is a finite set of inputs that tests ϕ against any processing function in Φ' .

Definition 5.4. Let Z be a DSXM having type Φ . Then a finite set $V = \{v_1, \dots, v_k\} \subseteq \Phi^*$ is called a *function cover* of Z if Φ can be written as $\Phi = \{\phi_1, \dots, \phi_k\}$ such that the following hold:

- $v_1 = \epsilon$ and $\phi_1 \in L_{A_Z}$;
- $\forall 2 \leq i \leq k, (v_i \in \{\phi_1, \dots, \phi_{i-1}\}^* \text{ and } v_i \phi_i \in L_{A_Z})$.

That is, v_i is a sequence containing only the functions $\phi_1, \dots, \phi_{i-1}$ that “accesses” ϕ_i . Therefore, V accesses every processing function in A_Z using sequences of functions that have already been accessed.

Example 5.1. For Z as in Example 3.1., $V = \{\epsilon, \phi_1, \phi_1\phi_2\}$ is a function cover of Z .

A function cover of Z exists if and only if for any proper subset Φ_0 of Φ , $L_Z \setminus \Phi_0^* \neq \emptyset$. This happens if A_Z is accessible and all processing functions in Φ are actually used as labels in A_Z (i.e. $\pi_2(\text{dom}(F)) = \Phi$), as it is always the case in practice.

Definition 5.5. Let $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $Z' = (\Sigma, \Gamma, Q', M, \Phi', F', q'_0, m_0)$ be two weak testing compatible DSXMs, $\Phi = \{\phi_1, \dots, \phi_k\}$ input-complete, $V = \{v_1, \dots, v_k\}$ a function cover of Z and t a test function of Z . Then $X_f = \cup_{i=1}^k t(v_i)\Sigma_{m_i}^i$ is called a *function test set of Z w.r.t. Φ'* if for $1 \leq i \leq k$, $\pi_2(|v_i| | (m_0, t(v_i))) = m_i$ and $\Sigma_{m_i}^i$ is a weak m_i test set of ϕ_i w.r.t. Φ' .

For simplicity, in the expression of X_f we used $t(v_i)$ instead of $\{t(v_i)\}$. A function test set of Z exists if a function cover of Z exists and Φ is input-complete.

The underlying idea behind the construction of a function test set is to access and test every processing function in A_Z using sequences of functions that have already been accessed and tested. Therefore, a function test set is used to test the processing functions of the DSXM specification against their implementations, as shown by the result below.

Lemma 5.1. Let Z and Z' be two weak testing compatible DSXMs having types Φ and Φ' , respectively, such that Φ is input-complete and output-distinguishable w.r.t. Φ' . If X_f is a function test set of Z w.r.t. Φ' and $f_Z \upharpoonright X_f = f_{Z'} \upharpoonright X_f$ then there exists an injective function $c : \Phi \longrightarrow \Phi'$ such that $\forall \phi \in \Phi, \phi \subseteq c(\phi)$.

Proof. Let $\Phi = \{\phi_1, \dots, \phi_k\}$, $V = \{v_1, \dots, v_k\}$ and $X_f = \cup_{i=1}^k t(v_i)\Sigma_{m_i}^i$ be as in Definition 5.5.. Let also $c : \Phi \longrightarrow \Phi'$ be as in Definition 5.2. For simplicity, we also denote by $c : \Phi^* \longrightarrow \Phi'^*$ the free-semigroup morphism induced by c .

We prove by induction on $1 \leq i \leq k$ the following statement: $\phi_i \subseteq c(\phi_i)$ and $c(v_i\phi_i) \in L_{A_{Z'}}$.

For $i = 1$ this is $\phi_1 \subseteq c(\phi_1)$ and $c(\phi_1) \in L_{A_{Z'}}$. Since $f_Z \upharpoonright \Sigma_{m_1}^1 = f_{Z'} \upharpoonright \Sigma_{m_1}^1$, where $m_1 = m_0$, $\exists \phi' \in \Phi'$ such that $\phi' \in L_{A_{Z'}}$ and $\forall \sigma \in \Sigma_{m_1}^1, \pi_1(\phi_1(m_1, \sigma)) = \pi_1(\phi'(m_1, \sigma))$. Since Φ is output-distinguishable w.r.t. Φ' , we have $\phi' = c(\phi_1)$, so $c(\phi_1) \in L_{A_{Z'}}$ and $\forall \sigma \in \Sigma_{m_1}^1, \pi_1(\phi_1(m_1, \sigma)) = \pi_1(c(\phi_1)(m_1, \sigma))$. Since $\Sigma_{m_1}^1$ is a weak m_1 test set of ϕ_1 w.r.t. Φ' , we have $\phi_1 \subseteq c(\phi_1)$.

Assume the statement true for $1 \leq j \leq i-1, i > 1$. Let $s_j = t(v_j)$. Since $\forall 1 \leq j \leq i, v_j \in \{\phi_1, \dots, \phi_{j-1}\}^*$ and $\phi_j \subseteq c(\phi_j)$, we have $\pi_1(|v_i| | (m_0, s_i)) = \pi_1(|c(v_i)| | (m_0, s_i))$ and $\pi_2(|v_i| | (m_0, s_i)) = \pi_2(|c(v_i)| | (m_0, s_i)) = m_i$.

From this and $f_Z \mid s_i \Sigma_{m_i}^i = f_{Z'} \mid s_i \Sigma_{m_i}^i$ it follows that $\exists \phi' \in \Phi'$ such that $c(v_i)\phi' \in L_{A_{Z'}}$ and $\forall \sigma_i \in \Sigma_{m_i}^i$, $\pi_1(\mid v_i \phi_i \mid (m_0, s_i \sigma_i)) = \pi_1(\mid c(v_i)\phi' \mid (m_0, s_i \sigma_i))$. Hence $\forall \sigma_i \in \Sigma_{m_i}^i$, $\pi_1(\phi_i(m_i, \sigma_i)) = \pi_1(\phi'(m_i, \sigma_i))$. Since Φ is output-distinguishable w.r.t. Φ' , we have $\phi' = c(\phi_i)$, so $c(v_i \phi_i) \in L_{A_{Z'}}$ and $\forall \sigma_i \in \Sigma_{m_i}^i$, $\pi_1(\phi_i(m_i, \sigma_i)) = \pi_1(c(\phi_i)(m_i, \sigma_i))$. Since $\Sigma_{m_i}^i$ is a weak m_i test set of ϕ_i w.r.t. Φ' , it follows that $\phi_i \subseteq c(\phi_i)$. \square

We can now prove the result we are after. Theorem 5.1. is the theoretical basis for the complete DSXM testing method.

Theorem 5.1. Let Z be a DSXM having type Φ input-complete and C a set of DSXMs weak testing compatible with Z having type Φ' such that Φ is output-distinguishable w.r.t. Φ' . If t is a test function of Z , $Y \subseteq \Phi^*$ is a test set of A_Z w.r.t. A_C^{-c} , where $A_C^{-c} = \{A_{Z'}^{-c} \mid Z' \in C\}$ and X_f is a function test set of Z w.r.t. Φ' , then $X' = t(Y) \cup X_f$ is a weak test set of Z w.r.t. C .

Proof. Let $Z' \in C$ such that $f_Z \mid X' = f_{Z'} \mid X'$. Then we have to prove that $f_Z \subseteq f_{Z'}$.

For simplicity, we use c to denote both the injective function $c : \Phi \rightarrow \Phi'$ and the free-semigroup morphism induced by it, $c : \Phi^* \rightarrow \Phi'^*$. From Lemma 5.1. it follows that $\forall \phi \in \Phi$, $\phi \subseteq c(\phi)$.

Let $v \in Y$. We prove that $v \in L_{A_Z}$ iff $c(v) \in L_{A_{Z'}}$.

“if”: Assume $v \in L_{A_Z}$. Then from $f_Z(t(v)) = f_{Z'}(t(v))$ it follows that $\exists v' \in \Phi'^*$ such that $\pi_1(\mid v \mid (m_0, t(v))) = \pi_1(\mid v' \mid (m_0, t(v)))$. Since Φ is output-distinguishable w.r.t. Φ' and $\forall \phi \in \Phi$, $\phi \subseteq c(\phi)$, we have $v' = c(v)$, so $c(v) \in L_{A_{Z'}}$.

“only if”: Assume $v \notin L_{A_Z}$ and $c(v) \in L_{A_{Z'}}$. Then let $v = \phi_1 \dots \phi_n$ and let $1 \leq k \leq n$ be such that $\phi_1 \dots \phi_{k-1} \in L_{A_Z}$ and $\phi_1 \dots \phi_k \notin L_{A_Z}$. Then $(m_0, t(v)) \in \text{dom}(\mid \phi_1 \dots \phi_k \mid)$. Since $\forall \phi \in \Phi$, $\phi \subseteq c(\phi)$, we have $(m_0, t(v)) \in \text{dom}(\mid c(\phi_1) \dots c(\phi_k) \mid)$. Since $c(v) \in L_{A_{Z'}}$, we have $c(\phi_1) \dots c(\phi_k) \in L_{A_{Z'}}$, so from $f_Z(t(v)) = f_{Z'}(t(v))$ it follows that $\exists v' \in \Phi'^*$ such that $v' \in L_{A_Z}$ and $\pi_1(\mid v' \mid (m_0, t(v))) = \pi_1(\mid c(\phi_1) \dots c(\phi_k) \mid (m_0, t(v)))$. Since Φ is output-distinguishable w.r.t. Φ' and $\forall \phi \in \Phi$, $\phi \subseteq c(\phi)$, we have $v' = \phi_1 \dots \phi_k = v$, so $v \in L_{A_Z}$, which is a contradiction.

Therefore $(v \in Y \text{ and } v \in L_{A_Z})$ iff $(v \in Y \text{ and } c(v) \in L_{A_{Z'}})$, hence $(v \in Y \text{ and } v \in L_{A_Z})$ iff $(v \in Y \text{ and } v \in L_{A_{Z'}}^{-c})$. Since Y is a test set of A_Z w.r.t. A_C^{-c} , it follows that $v \in L_{A_Z}$ iff $v \in L_{A_{Z'}}^{-c}$. Therefore, $v \in L_{A_Z}$ iff $c(v) \in L_{A_{Z'}}$. From this and $\forall \phi \in \Phi$, $\phi \subseteq c(\phi)$ it follows that $f_Z \subseteq f_{Z'}$. \square

If the specification Z is completely specified, then X' establishes functional equivalence of specification and implementation rather than functional inclusion.

Corollary 5.1. In the conditions of Theorem 5.1., if Z is completely specified then $X' = t(Y) \cup X_f$ is a test set of Z w.r.t. C .

Proof. Follows from Theorem 5.1. and the fact that f_Z is a total function. \square

6. The complete DSXM testing method

We can now use Theorem 5.1. and Theorem 2.3. to generate weak test sets for a DSXM specification.

6.1. Pre-requisites

The method works under the following assumptions:

1. The specification Z is a DSXM whose associated FA is minimal.
2. The type Φ of the specification is input-complete and output-distinguishable.
3. The implementation can be modelled by a DSXM Z' such that Φ is output-distinguishable w.r.t. Φ' .
4. For any processing function $\phi \in \Phi$ and memory value $m \in M$, a weak m test set of ϕ w.r.t. Φ' can be constructed.
5. The number of states in Z' is bounded by an integer m , which is larger than or equal to the number n of states in Z .

Of these assumptions, the first two lie within the capability of the designer. This can arrange for the associated FA of the X-machine specification to be minimal; standard techniques from automata theory are available [Eil74]. The design for test conditions (i.e. input-completeness and output-distinguishability) can be easily introduced in the definitions of the processing functions by using extra input and output symbols; a very simple algorithm is given in [HoI98]; the extra inputs and outputs can be filtered out once the system has passed testing.

The third condition is satisfied if the implementation of each processing function can be identified by examining outputs. This is normally true if the specified processing functions are output-distinguishable, provided that all occurrences of a processing function are implemented in the same way or by the same piece of code. When it is not possible to ensure that two or more occurrences of a processing function are implemented in the same way, the specification can be modified such that each occurrence will become a distinct function; this can be achieved by using extra outputs that can be filtered out once the testing has been completed. Note that, unlike the DSXM integration testing method, here it is not assumed that the processing functions are *correctly* implemented, but only that their implementations are present and can be distinguished. Furthermore, the system implementation may contain additional components (i.e. implementations of extra processing functions) providing that the output-distinguishability property is preserved.

A weak m test set of a processing function ϕ can be constructed using the same method if ϕ is expressible as the computation of another, simpler, stream X-machine. Alternatively, other functional testing approaches (e.g. the category partition method or a variant [OsB89]) can be used if ϕ is a function that carries out relatively simple tasks on data structures (i.e. inserting and removing items from registers, stacks, files, etc.). Note that, unlike the DSXM integration testing method, here it is not assumed that the processing functions can be tested in isolation from the rest of the system and their implementation proved to be correct before the integration testing can begin.

In practice, m is not usually much larger than n , for especially sensitive applications one can make very pessimistic assumptions about m at the cost of a larger test set.

6.2. Generation of the test set

Under these conditions, a weak test set is $X'_{m-n} = t(Y_{m-n}) \cup X_f$, where

- t is a test function of Z ,
- $Y_{m-n} = P(\Phi^{m-n} \cup \dots \cup \{\epsilon\})(W \cup \{\epsilon\})$,
- P is a transition cover of A_Z and W is a characterisation set of A_Z ,
- $X_f = \bigcup_{i=1}^k t(v_i)\Sigma_{m_i}^i$ is a function test set of Z w.r.t. Φ' , where $V = \{v_1, \dots, v_k\}$ is a function cover of Z , $\Phi = \{\phi_1, \dots, \phi_k\}$ and for $1 \leq i \leq k$, $\pi_2(|v_i| (m_0, t(v_i))) = m_i$ and $\Sigma_{m_i}^i$ is a weak m_i test set of ϕ_i w.r.t. Φ' .

The following example illustrates the construction of X'_{m-n} for Z as in Example 3.1. and $m - n = 1$.

Example 6.1. We assume that for any memory value $m \in M$ we have $\Sigma_m^1 = \{a\}$, $\Sigma_m^2 = \{a\}$, $\Sigma_m^3 = \{b\}$. S , P , W , Y_1 and X_1 are as in Example 4.2..

$$\begin{aligned} V &= \{\epsilon, \phi_1, \phi_1\phi_2\}, \\ X_f &= \Sigma_0^1 \cup \{a\}\Sigma_0^2 \cup \{aa\}\Sigma_0^3 = \{a, aa, aab\}, \\ X'_1 &= X_1 \cup X_f. \end{aligned}$$

6.3. Complexity of the method

The final question that needs to be addressed is concerned with the practicality of the method. That is, how complex is the test generation algorithm and what is the size of the test set generated?

In what follows, for a finite set of sequences $B \subseteq A^*$, $|B| = \sum_{b \in B} \text{length}(b)$ denotes the total length of all sequences in B .

According to [Cho78], $\text{card}(Y_{m-n}) \leq n^2 k^{m-n+1}$ and $|Y_{m-n}| \leq n^2 m k^{m-n+1}$, where $k = \text{card}(\Phi)$. Since the size of $X_{m-n} = t(Y_{m-n})$ cannot exceed that of Y_{m-n} , $\text{card}(X_{m-n}) \leq n^2 k^{m-n+1}$ and $|X_{m-n}| \leq n^2 m k^{m-n+1}$. On the other hand, $v_1 = \epsilon$ and it is easy to see that for $2 \leq i \leq k$, $\text{length}(v_i) \leq n - 1$, so $|\{v_1, \dots, v_k\}| \leq (n - 1)(k - 1)$. If the number of elements of any weak m_i test set $\Sigma_{m_i}^i$ of a processing function is at most r , then $\text{card}(X_f) \leq kr$ and $|X_f| \leq (n - 1)(k - 1)r + kr \leq nkr$. Therefore $\text{card}(X') \leq n^2 k^{m-n+1} + kr$ and $|X'| \leq n^2 m k^{m-n+1} + nkr$. In particular, for $m = n$, the respective bounds are $n^2 k + kr$ and $n^3 k + nkr$. Note that these bounds refer to the worst case, in an average case, the size of the test set is much lower.

According to [Cho78], the amount of work required to construct Y_{m-n} is n^2k . The set $X_{m-n} = t(Y_{m-n})$ is generated by first constructing the test set Y_{m-n} and then applying the test function to each sequence in Y_{m-n} . The complexity of the former is n^2k , the complexity of the latter is proportional to the complexity of the processing functions, the size of the input alphabet and the total length of Y_{m-n} . Therefore, if the complexity of each processing function is at most C and $\text{Card}(\Sigma) = p$ then the complexity of constructing X_{m-n} is $n^2k + Cpn^2mk^{m-n+1} \approx Cpn^2mk^{m-n+1}$. The set X_f is generated by first constructing a function cover $V = \{v_1, \dots, v_k\}$, then applying the test function to each sequence in V and then generating the weak m_i test sets of the processing functions. The amount of work required to construct V is proportional to nk . The complexity of applying the test function to V is proportional to the complexity of the processing functions, the size of the input alphabet and the total length of V , so this is $Cpnk$. Therefore, if the complexity of generating a weak m_i test set $\Sigma_{m_i}^i$ of a processing function is at most D then the complexity of constructing X_f is $nk + Cpnk + kD \approx Cpnk + kD$. Therefore, the total complexity of the method is $Cpn^2mk^{m-n+1} + Cpnk + kD \approx Cpn^2mk^{m-n+1} + kD$.

If the processing functions and their weak m_i test sets are computable by some algorithms, then the generation of the test set can be automated. Clearly, the method will have to be supported by automated systems and suitable tools which do not yet exist.

7. Conclusions

The complete DSXM testing method presented here generalises the existing DSXM integration testing method. It no longer requires the implementations of the processing functions to be proved correct before the actual testing can take place. Instead, the testing of the processing functions is performed along with the integration testing. This is an important advance since, in many situations, the implementations of the processing functions are not separate units of code (functions, procedures, etc.) that can be tested in isolation from the rest of the system.

Consequently, the test set generated by the method is made up of two components: a set for testing the processing functions (called a function test set) and a set for testing their integration (i.e. the test set generated by the DSXM integration testing method).

Further work involves the application of the method to real case studies and the development of automated tools to support it as well as the generalisation of the method to *non-deterministic* stream X-machine specifications, where each transition is associated with a processing *relation* rather than with a function and the domains of the relations that emerge from the same state may overlap.

References

- [ABC02] Aguado J, Bălănescu T, Cowling T, Gheorghe M, Holcombe M, Ipate F (2002) P systems with replicated rewriting and stream X-machines (Eilenberg machines). *Fundamenta Informaticae* 49(1–3):17–33
- [BCG09] Bălănescu T, Cowling T, Georgescu H, Gheorghe M, Holcombe M, Vertan C (1999) Communicating stream X-machines are no more than X-machines. *J Universal Comput Sci* 5:494–507
- [BGH00] Bălănescu T, Gheorghe M, Holcombe M (2000) Deterministic stream X-machines based on grammar systems. In: Martin-Vide C, Mitrana V (eds) *Words, sequences, grammars, languages: where biology, computer science, linguistics and mathematics meet*, vol 1. Kluwer, Dordrecht, pp 13–23
- [Bal01] Bălănescu T (2000) Generalized stream X machines with output delimited type. *Formal Aspects Comput* 12:473–484
- [BGI03] Bălănescu T, Gheorghe M, Ipate F, Holcombe M (2003) Formal black box testing for partially specified deterministic finite state machines. *Foundations Comput Decis Syst* 28(1):17–28
- [BGH01] Bălănescu T, Gheorghe M, Holcombe M, Ipate F (2001) Testing collaborative agents defined as stream X-machines. In: *Advances in artificial life, proceedings of the 6th European conference ECAL, Prague, 10–14 September 2001*. Springer, Berlin Heidelberg New York, pp 296–305
- [BWW96] Barnard J, Whitworth J, Woodward M (1996) Communicating X-machines. *Inf Software Technol* 38:401–407
- [ChK93] Cheng K-T, Krishnakumar AS (1993) Automatic functional test generation using the extended finite state machine model. In: *Proceedings of the 30th design automation conference dallas, 14–18 June 1993*. ACM Press, New Orleans, pp 86–91
- [Cho78] Chow TS (1978) Testing software design modelled by finite state machines. *IEEE Trans Software Eng* 4(3):178–187
- [CHV00] Cowling A, Georgescu H, Vertan C (2000) A structured way to use channels for communication in X-machine systems. *Formal Aspects Comput* 12(6):458–500
- [Eil74] Eilenberg S (1994) *Automata, languages and machines*, vol A. Academic, New York
- [FHI95] Fairtlough M, Holcombe M, Ipate F, Jordan C, Laycock G, Duan Z (1995) Using an X-machine to model a video cassette recorder. *Curr Issues Electron Model* 3:141–161
- [GeV00] Georgescu H, Vertan C (2000) A new approach to communicating X-machines. *J Universal Comput Sci* 6(5):490–502
- [Ghe00] Gheorghe M (2000) Generalized stream X-machines and cooperating distributed grammar systems. *Formal Aspects Comput* 12(6):459–472

- [HiH00] Hierons RM, Harman M (2000) Testing conformance to a quasi-non-deterministic stream X-machine. *Formal Aspects Comput* 12(6):423–442
- [Hol88] Holcombe M (1988) X-machines as a basis for dynamic system specification. *Software Eng J* 3:69–76
- [MIG95] Holcombe M, Ipaté F, Grondoudis A (1995) Complete functional testing of safety-critical systems. In: *Proceedings of the 2nd IFAC workshop on safety and reliability in emerging control technologies Daytona Beach, 1–3 November 1995*. Elsevier, Oxford, pp199–204
- [HoI98] Holcombe M, Ipaté F (1998) *Correct systems: building a business process solution*. Springer, Berlin Heidelberg New York
- [IpH96] Ipaté F, Holcombe M (1996) Another look at computability. *Informatica* 20:359–372
- [IpH97] Ipaté F, Holcombe M (1997) An integration testing method that is proved to find all faults. *Int J Comput Math* 63:159–178
- [IpH98a] Ipaté F, Holcombe M (1998) A method for refining and testing generalized machine specifications. *Int J Comput Math* 68:197–219
- [IpH98b] Ipaté F, Holcombe M (1998) Specification and testing using generalized machines: a presentation and a case study. *Software Test Verification Reliability* 8:61–81
- [IpH00] Ipaté F, Holcombe M (2000) Generating test sequences from non-deterministic generalized stream X-machines. *Formal Aspects Comput* 12(6):443–458
- [IpH02a] Ipaté F, Holcombe M (2002) An integrated refinement and testing method for stream X-machines. *Applicable Algebra Eng Commun Comput* 13(2):67–91
- [IpH02b] Ipaté F, Holcombe M (2002) Testing conditions for communicating stream X-machine systems. *Formal Aspects Comput* 13(6):431–446
- [Ipa03] Ipaté F (2003) On the minimality of stream X-machines. *Comput J* 46(3)
- [KeK00] Kefalas P, Kapeti E (2000) A design language and tool for X-machine specification. In: Fotadis DI, Nikolopoulos SD (eds) *Advances in informatics I*, world scientific, Athens, pp 134–145
- [KEK00] Kehris E, Eleftherakis G, Kefalas P (2000) Using X-machines to model and test discrete event simulation programs. In: Mastrokakis N (ed) *systems and control: theory and applications*, world scientific and engineering. Society Press, Athens, pp 163–171
- [LeY96] Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines – A survey. *Proc IEEE* 84(8):1090–1123
- [OsB89] Ostrand TJ, Balcer MJ (1989) The category-partition method for specifying and generating functional tests. *Commun ACM* 31(6):667–686
- [WaL93] Wang C-J, Liu MT (1993) Generating test cases for EFSM with given fault models. In: *Proceedings of IEEE INFOCOM '93 vol 2, San Francisco, 28 March – 1 April 1993*. IEEE, pp 774–781

Received 11 June 2003

Accepted in revised form 27 February 2004 by D.A.Duce

Published online 2 June 2004