

# Testing methods for X-machines: a review

K. Bogdanov<sup>1</sup>, M. Holcombe<sup>1</sup>, F. Ipate<sup>2</sup>, L. Seed<sup>3</sup>, S. Vanak<sup>1</sup>

<sup>1</sup> Department of Computer Science, The University of Sheffield, Regent Court, 211 Portobello St., Sheffield S1 4DP, UK

<sup>2</sup> Department of Computer Science and Mathematics, University of Pitesti, Str Targu din Vale 1, 0300 Pitesti, Romania

<sup>3</sup> Department of Electronic & Electrical Engineering, The University of Sheffield, Mappin St., Sheffield S1 3JD, UK

**Abstract.** The X-machine testing method has been developed as an application of the W-method to testing the control structure of an implementation, against a specification. The method was proven to demonstrate the equivalence of the behaviour of the two, subject to a number of conditions both a specification and an implementation are expected to satisfy, such as (1) determinism of the two and (2) that functions labelling arcs on a transition diagram of a specification control structure have been tested in advance. Since the original publication of the testing method, a number of extensions have been published, removing the restrictions mentioned above. This paper surveys the extensions of the X-machine testing method, for (1) testing of functions together with testing of a transition diagram, (2) equivalence testing of a non-deterministic implementation against a non-deterministic specification, (3) conformance testing of a deterministic implementation against a non-deterministic specification and (4) equivalence testing of a system of concurrently executing and communicating X-machines, against a specification.

**Keywords:** Software testing; Test set generation; X-machines; Finite state machines

## 1. Introduction

Formal methods have played an important role in the development of techniques for the specification and verification of computer systems over the last 20 years. Although there have been a great number of advances in research in this field, the uptake in industry has been disappointing. Part of the problem is that the average software engineer finds many of the notations difficult and unintuitive. Furthermore, real software development projects rarely fit with the ‘waterfall’ model that is the basis for much of the formal methods currently proposed. It is very rare for a large system to have stable requirements and thus the concept of a large formal model which can be analysed and used as the basis for significant effort in, for example, the formal verification of the system is impractical and uneconomic.

There is also scepticism about the value of much of the effort expended on formal verification activities that require substantial expertise and do not scale well. In any event, the need to model the operational environment accurately is almost impossible and therefore weakens any formal conclusions that can be made about the properties and behaviour of, what is likely to be a partially modelled, operating system.

Whatever else happens in a software development project or in a hardware design project there will be extensive testing activity. Thus, the implementation will be executed and ‘used’ in a realistic environment and be subjected

to a series of tests or experiments which will provide information about the fitness for purpose of the developed system. Testing is limited to a series of finite experiments and, thus, the conclusions that can be drawn from it are potentially limited in that they cannot be generalised to cover every possible eventuality. Some formal methods can make a useful contribution to testing and this paper surveys one of the most effective modern approaches.

A number of authors have investigated the issue of generating test sets from formal specifications [Ber91, DiF93, Tre96, Vas73, Cho78] – and there has been some study of the properties of such test sets, for example optimising the number or lengths of test sequences [Hie97b, Hie98, HU02, IU99] – and related issues.

The main point about X-machine testing, the type of test generation technique discussed here, however, is to establish, not only efficient test sets from a specification but also to relate the test process to a hierarchical decomposition of the implementation in order to make the management of the test process more convenient.

In this paper we summarise the main achievements of this approach. X-machine based testing has been developed in a number of directions and work on testing non-deterministic, communicating and hardware systems has been successful in addressing both of the points made above. Moreover, a number of case studies have been carried out to establish if the approach is actually practical. Thus, a number of systems have been specified as X-machines and the tests produced by the approach used to validate the implementations. In one of these, a comparison was made with the current test process in a leading design house using their current design projects as the basis for a detailed comparison concerning the cost and effectiveness of the two methods. This is described in Sect. 4.

Before that we will consider some further issues relating to testing from formal specifications and set the scene for the rest of the paper. The basis of the method is to utilise the theory of testing devised for finite state machines and to generalise it to the stream X-machine situation – thus enabling the method to be more convincing for a wide variety of practical applications as well as providing mechanisms that can control state explosion problems. Before we do this, let us ask what we might want from a testing method,

- applicability – it is applicable to the systems we are analysing;
- efficient – the test sets are neither too numerous nor the test sequences too long;
- effective – the test method finds faults well;
- usable – it should be based on user friendly specification methods and the test set construction should be easy to carry out;
- supported – the method should be as automated as possible. The hard work is in constructing and validating the specification – testing should be easy;
- analysable – we can say something about the correctness of the system after testing is complete.

Let us look at each of these issues in turn. Unlike many other methods, the X-machine testing method appears to score well on all these points. The X-machine method is particularly good on the last point – and for critical systems design – perhaps the most important one. All functional testing must start from a specification or a requirement of some sort. For critical systems this model should encapsulate the most important constraints of the system from a safety point of view. These relate to:

- dynamic control space;
- data processing details;
- timing;
- environmental interfaces;
- communication between concurrent processes.

As we have noted, above, the stability of the specification is likely to be low and so there must be mechanisms for specification revision and development that allows for changes in the specification to be dealt with. Unfortunately, most of the time, data plays a significant part and the natural generalisations of finite state machines to incorporate data modelling are, essentially threefold. Extended state machines and abstract state machines [Gur00] have the traditional state space diagram but the arcs are labelled by conditional logical expressions. This allows for the semantics of higher order actions and events to be captured in a formal sense. However, they are poor at providing a hierarchical structure to the model since there is only one level of state architecture. Statecharts [HaN96, Omg03] do provide a hierarchical approach to encapsulating sub-machines within states thus providing a mechanism for the treatment of large state spaces in an encapsulating manner. The problem with statecharts is, however, the many different types of semantics that have been defined [Bee94]. This is partly the result of allowing many types of design operation within the model, for example, inter level transitions can

create many difficulties in interpretation. Transitions of statecharts only provide limited scope for abstraction and modelling subprocesses.

X-machines, on the other hand, provide a number of advantages. The model is fully general (Turing machines are special cases of general X-machines) as well as being extremely expressive. The labels of the arcs in the state diagram are functions that operate upon defined data sets, these may be inputs, internal memory variables, outputs etc. This provides a key method for hierarchical definition since these functions, themselves, can be the functions computed by other, simpler, X-machines.

The development of refinement theories and, more recently, investigations into change transformation driven by requirements change offer powerful advantages, especially if the test set generation algorithms can be related to the change transformation in a way that the refinements can [ThH03].

In test generation using extended finite state machines or statecharts the mechanisms used to rely on the ‘flattening’ of the state space in order to apply finite state machine testing methods. In other words, every different combination of values of variables used in a system under test is included in the state space for test generation. For this reason, what has been gained by abstracting away some of the state space is lost when it comes to the test set generation. The situation with X-machines is different. The fundamental approach, using, for example, Chow’s technique, preserves the hierarchical structure of the model and provides an extremely powerful means of completely testing an implementation subject to a number of conditions. The adaptation of the X-machine testing method to statecharts is mentioned in the conclusions of this paper.

The essence of X-machine testing is the following:

The specification is defined as an X-machine – there are a number of different types, the deterministic Stream X-machine is the basic model but there are a number of more general ones which will be described below. The model is adjusted to ensure that it satisfies two ‘design for test’ conditions, namely controllability (input-completeness, Definition 3.5) and observability (output-distinguishability, Definition 3.4) and this can always be done using some general algorithms. A more general form of these conditions is described in Sect. 6. An estimate of the size of the state space in the implementation is also required; this affects the size of the test set (the less certainty about the number of extra states in the implementation there is, the greater the test set has to be). Finally, the test method assumes that the lower level testing has been done successfully. This means that the individual functions on the arcs of the machine are correct. These, as has been noted, could be X-machines themselves and thus the technique reduces to the testing of these and so on, until we reach a level where the basic functions on the arcs of the machines are dependable and we can assume their correctness. Thus we have a hierarchical method of testing that follows the hierarchical decomposition of the model. A very powerful attribute since it provides, automatically, an effective mechanism for managing the test process and aligning it to modern software and hardware engineering techniques such as component based design whereby ‘basic functions’ can be reused and the test problem becomes one of testing the integration of a number of dependable components.

In the following sections we will examine the basic theory of the technique, look at the issue of non-determinism and examine an industrial case study which compared the method with the current state of the art in a high profile company. In this investigation the X-machine method was applied to the company’s current designs and some exceptional results were obtained.

## 2. Background: finite automata and stream X-machines

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  $A$ .  $\epsilon$  denotes the empty sequence. For  $a, b \in A^*$ ,  $ab$  denotes the concatenation of sequences  $a$  and  $b$ . Here  $a^n$  is defined by  $a^0 = \epsilon$  and  $a^n = a^{n-1}a$  for  $n \geq 1$ . Given two sets of sequences  $U, V \subseteq A^*$ ,  $UV = \{ab \mid a \in U, b \in V\}$ ;  $U^n$  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  $length(\epsilon) = 0$ ). In testing, one might often like to find out the total length of a test set, hence, for a finite set of sequences  $B \subseteq A^*$ ,  $length(B) = \sum_{b \in B} length(b)$  denotes the total length of all sequences in  $B$ .

Given a sequence  $a \in A^*$  we say that  $b \in A^*$  is a *prefix* of  $a$  if there exists  $c \in A^*$  such that  $a = bc$ . The set of all prefixes of  $a$  is denoted by  $pref(a)$ , i.e.  $pref(a) = \{b \in A^* \mid \exists c \in A^* \text{ such that } a = bc\}$ . For  $U \subseteq A^*$ ,  $pref(U) = \cup_{a \in U} pref(a)$ . For a relation (or partial function)  $f : A \longleftrightarrow B$ ,  $dom(f)$  denotes the domain of  $f$  and  $Im(f)$  the image of  $f$ . For  $a \notin dom(f)$ ,  $f(a) = \emptyset$ . For  $U \subseteq A$ ,  $f(U) = \cup_{a \in U} f(a)$ . For  $U \subseteq A$ ,  $f \mid U$  denotes the restriction of  $f$  to  $U$ , i.e.  $f \mid U : U \longleftrightarrow B$  is defined by  $f \mid U(a) = f(a), \forall a \in U$ . Given two relations (or partial functions)  $f, g : A \longleftrightarrow B$ , we use  $f \subseteq g$  to denote that  $f(a) \subseteq g(a), \forall a \in A$ . For  $n$  sets  $A_1, \dots, A_n$ ,

$\pi_i : A_1 \times \cdots \times A_n \longrightarrow A_i$  denotes the projection function, for  $1 \leq i \leq n$ . For a finite set  $A$ ,  $\text{card}(A)$  denotes the number of elements in  $A$ .

## 2.1. Finite automata

This subsection 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  $(\Sigma, Q, F, I, T)$ , where:

- $\Sigma$  is the finite *input alphabet*;
- $Q$  is the finite *set of states*;
- $F$  is the (partial) *next state function*,  $F : Q \times \Sigma \longrightarrow 2^Q$ ;
- $I$  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  $\sigma$  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 this paper, only FAs with all states terminal ( $T = Q$ ) are considered, hence an FA will be denoted by a tuple  $(\Sigma, Q, F, q_0)$ . Sections. 3 and 5 only consider deterministic FAs with non-determinism deferred to Sect. 7.

**Definition 2.3** The next state function can be extended to a (partial) function  $F^* : Q \times \Sigma^* \longrightarrow Q$  defined by  $F^*(q, \epsilon) = q, \forall q \in Q$  and  $F^*(q, s\sigma) = F(F^*(q, s), \sigma), \forall q \in Q, s \in \Sigma^*, \sigma \in \Sigma$ .

**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  $A'$  that does not conform to  $A$ .

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

It is important to note that an implementation has to feature a reliable reset which can be used to reset it after every sequence from a test set. This is important because after every test sequence, the state entered by a potentially faulty implementation is not known and a reset brings the system into a known state. Every test sequence is implicitly assumed in this paper to contain a reset. Complete testing without reset is possible [HU02, IU99], but is not considered in this paper.

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

There is 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* [Vas73, Cho78, BGI03]: the difference between the number of states of the implementation and that of the specification has to be at most  $k$ , a positive integer estimated by the tester. The *W-method* was first presented in [Vas73, Cho78], in the context of input/output finite state machines (i.e. for which  $F$  is of the form  $F : Q \times \Sigma \longrightarrow 2^{Q \times \Gamma}$ , where  $\Gamma$  is the output alphabet) that are *completely defined* (i.e.  $\emptyset \notin \text{Im}(F)$ ) 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. It is assumed that both a specification and an implementation ignore inputs for which no transitions are defined and that all states are distinguishable.

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 3.2. 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  $\Sigma$ ,  $n$  the number of states of  $A$ ,  $m \geq n$  and  $C_m$  the set of deterministic FAs having input alphabet  $\Sigma$  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.  $C_m$ .

It is assumed in this paper that a tester applies a test sequence as a whole and subsequently checks an output sequence produced by an implementation rather than applying an input sequence symbol-by-symbol and observing output symbols as they come out. This can often happen if there is a buffer between a tester and an implementation under test. If, for example, an implementation responds with a sequence  $x$  to an input sequence  $ab$ , a tester cannot tell whether an output of  $x$  was produced in response to the former or the latter element of the input sequence. The addition of  $\{\epsilon\}$  to the *W* set makes it possible to test incomplete implementations (used for test generation in Sects. 3,5); an alternative is to use a rather larger test set of  $\text{pref}(Y_{m-n})$  (used in Sect. 7). A rather more general case is described in [PY02].

There are variants of the *W-method*, such as the partial *W-method* (*Wp-method*) [FBK91] and the harmonized state identification method (*HSI-method*) [LuPB94], which can both reduce the size of the test set at the expense of a more complex generation algorithm. There are also alternatives to the mentioned methods, such as *Unique Input-Output* (UIO), UIOv and other methods, a number of which are reviewed in [RDT95]. Some of these methods, such as UIO, do not guarantee complete fault detection by testing. The general *W-method* is, however, sufficient for the purpose of this paper, so its variants and other methods will not be presented here.

## 2.2. Stream X-machines

In this subsection 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 *relations* (often *partial functions*) that operate on a *basic data*

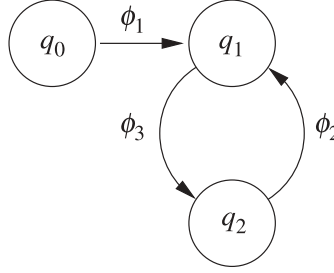


Fig. 1. The state transition diagram of an X-machine

set  $X$ . The set of these relations,  $\Phi$ , 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  $q_0$  and each edge is labelled by a relation:  $\phi_1$ ,  $\phi_2$ , etc. Sequences of relations are thus derived from each path in the state space and these may be composed to produce a relation that may be defined on the data state. This is then applied to the value  $x$  providing that the composed relation is defined on  $x$ . This then gives a new value,  $x \in X$  (if there is more than one such  $x$  then one will be picked in a non-deterministic fashion) for the data state and a new control state.

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 2.13** A *stream X-Machine* (SXM for short) is a tuple

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

where:

- $\Sigma$  and  $\Gamma$  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*;
- $\Phi$  is the *type* of  $Z$ , a finite set of distinct *processing relations* that the machine can use; a processing relation is a non-empty relation of the form  $\phi : M \times \Sigma \longleftrightarrow \Gamma \times M$ ; often  $\Phi$  is a set of (partial) functions.
- $F$  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 relations have the form  $\phi : M \times \Sigma \longleftrightarrow \Gamma \times M$ , i.e. each such relation will read an input symbol, discard it and produce one or more output symbols 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 relations from the type  $\Phi$ . The automaton  $A_Z = (\Phi, Q, F, I, T)$  over the alphabet  $\Phi$  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  $\phi$  is called an *arc* of  $Z$  from  $q$  to  $q'$ , denoted  $\phi : q \rightarrow q'$ .

A completely defined 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 2.14** A SXM  $Z$  is called *completely defined* if  $\forall q \in Q, m \in M, \sigma \in \Sigma, \exists \phi \in \Phi$  such that  $((m, \sigma) \in \text{dom}(\phi)$  and  $(q, \phi) \in \text{dom}(F)$ ).

A SXM which is not complete is assumed to ignore an input which does not cause any transition, by remaining in the same state with an unchanged value of memory.

**Definition 2.15** Given a sequence  $p \in \Phi^*$ ,  $p$  induces the relation

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

defined as follows:

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

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 relations in the sequence  $p$ .

It is easy to see that if  $\Phi$  is a set of (partial) functions rather than relations then  $\|p\|$  is also a (partial) function.

In general, a SXM may be *nondeterministic*, in the sense that the application of an input  $\sigma \in \Sigma$  in a state  $q \in Q$  for a memory value  $m \in M$  may trigger more than one transition.

A *deterministic* SXM (DSXM for short) is a SXM for which its associated automaton is deterministic,  $\Phi$  is a set of partial functions and any two processing functions that emerge from the same state have non-intersecting domains.

**Definition 2.16** A SXM is *deterministic* if the following hold:

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

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

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

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

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

It is easy to see that in a DSXM there is at most one possible transition for any triplet  $q \in Q, m \in M, \sigma \in \Sigma$ . Also note that if a deterministic SXM is completely defined then there is exactly one transition for any triplet  $q \in Q, m \in M, \sigma \in \Sigma$ .

The correspondence between the input sequence applied to the machine and the output produced gives rise to the *relation computed* by the machine, as defined next. In general, a (nondeterministic) SXM computes a relation since the application of an input sequence may produce more than one output sequence.

**Definition 2.17** Given a SXM  $Z$ , the relation  $f_Z : \Sigma^* \longleftrightarrow \Gamma^*$  defined by:

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

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

If  $Z$  is a DSXM, then  $f_Z$  is a (partial) function rather than a relation. The converse is not true, since a SXM computing a deterministic function will not necessarily comply with Definition 2.16. This can be illustrated by a SXM with  $\Sigma = \Gamma = \{0, 1\}$ ,  $M = \emptyset$  and a transition diagram containing a few states and transitions between them, all labelled with the same function  $\phi(m, 1) = (0, m)$ . Regardless of the transition diagram, such a SXM computes the same partial function  $f_Z$ , defined for all sequences  $\sigma = \{1\}^n, n > 0$  and producing  $\{0\}^n$ . Furthermore, if  $Z$  is a completely defined DSXM then  $f_Z$  is a total function.

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 has to be done in order to ensure that the output produced by the machine can be viewed in any of its states; additionally, the testing method does not test to ensure that specific implementation states are terminal.

Thus, in what follows, we will refer to *deterministic SXMs* 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 2.1** 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. 1 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.

### 3. DSXM integration testing

This section presents the DSXM integration testing method [IpH97, HoI98]. This method 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.

#### 3.1. Theoretical basis

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 3.1** Let  $Z$  be a DSXM and  $C$  a set of DSXMs having the same input alphabet ( $\Sigma$ ) and output alphabet ( $\Gamma$ ) 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 3.2** 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 Sect. 1, the implementation will be assumed to have the same type as the specification.

**Definition 3.3** Two weak testing compatible DSXMs are called *testing compatible* if they have identical types.

The basic idea of the method is to translate test sets of the associated FA into test sets of the DSXM specification. This works if the DSXM specification satisfies two conditions: input-completeness and output-distinguishability.

**Definition 3.4**  $\Phi$  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 3.5**  $\Phi$  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 2.1,  $\Phi$  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]. 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. Without the described conditions, it would be extremely difficult to test a system properly using the testing method being described. For instance, if a method generates a particular sequence of functions to execute and there is no corresponding sequence of inputs to attempt this sequence, such a sequence cannot be executed. As a consequence, some faults of the implementation may go undetected. There is a number of ways this problem can be dealt with, such as the following:

- If an X-machine specification is being built before an implementation, it can be possible to add special test inputs and outputs to functions of the X-machine. A very simple algorithm for doing this is described in [HoI98]. These extra inputs and outputs can be filtered out once the system has passed testing. After implementation, test inputs will ensure that any function can be forced and outputs will provide an indication of the function which is taken by an implementation. When an implementation already exists, it may be retrofitted with those extra inputs in order for testing to be carried out; if it is not possible, the alternatives below have to be considered.

- If an implementation cannot be instrumented to satisfy the design for test conditions, it could be possible to replace these conditions with alternatives. For instance, if a system contains a singleton output alphabet, then whenever an output is observed by a tester it indicates that a transition was taken by the system under test. For such a system, one has to strengthen the input-completeness condition so that every function responds only to a specific input. This way, if an implementation is supplied with some input and produces an output, it is known that the function corresponding to the supplied input fired. An alternative approach has been applied to the case studies reported in Sect. 4, where input-completeness was not satisfied, but all internal variables were observable. In this case, the system under test can be driven through sequences of functions until the value of memory and the state of an X-machine under test are the expected ones.
- The aim of the X-machine testing described in this section is to check the equivalence between a specification and an implementation; this paper also describes conformance testing (Sect. 8) for a specific conformance relation. In the situation where it is not possible to satisfy the design for test conditions, one may consider identifying the appropriate conformance relation and adapting the testing method to demonstrate such a relation by testing.

We also need a mechanism, called a test function, that translates sequences of processing functions into sequences of inputs.

**Definition 3.6** Let  $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$  be a DSXM having type  $\Phi$  input-complete. Then a function  $t : \Phi^* \rightarrow \Sigma^*$  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  $\phi_{i+1}$ , the function that follows after this prefix.

Note that since  $\Phi$  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 3.1** The following values illustrate the construction of a test function for  $Z$  as in Example 2.1:

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

We can assemble all these into the following results which is the theoretical basis for DSXM integration testing.

**Theorem 3.1** [IpH97, HoI98] Let  $Z$  be a DSXM having type  $\Phi$  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.  $C$ .

### 3.2. Pre-requisites

The method works under the following assumptions:

1. The specification  $Z$  is a DSXM whose associated FA is minimal.
2. The type  $\Phi$  of the specification is input-complete and output-distinguishable.
3. The implementation can be modelled by a DSXM  $Z'$  having the same type (processing functions) as  $Z$  (testing compatibility, Definition 3.3)
4. The number of states in  $Z'$  is bounded by an integer  $m$  which is larger than or equal to the number of states  $n$  of the DSXM specification.

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 discussion of the design for test conditions (i.e. input-completeness and output-distinguishability) can be found after Definition 3.5.

The third condition is the most problematic. In practice, this means that the basic processing functions have to be shown to be correct and this is done with a separate testing process depending on their nature and complexity. The same method can be applied to test the basic processing functions if they are expressible as the computations of other, simpler X-machines. Alternatively, other testing approaches (e.g. the category partition method or a variant [OsB89]) can be used to test functions that carry out simple tasks on data structures (i.e. inserting and removing items from registers, stacks, files). Testing of functions additionally ensures that input and output alphabets of functions of an implementation are the same as those of a specification. If the basic processing functions are tried and tested with a long history of successful use (i.e. standard procedures, modules or objects from a library) then their individual testing could perhaps be assumed done. However, the bottom line is that these processing functions will have to be implemented as separate units of code (functions, procedures, methods, etc.) that can be tested in isolation from the rest of the system.

For the fourth condition,  $m$  can often be expected not to be much larger than  $n$ , since for a correct implementation, every extra state corresponds to a redundant state and hence the amount of redundancy can be expected to be related to a misunderstanding between a designer and an implementer. In case of a serious misunderstanding, not only numerous extra states but also incorrectly behaving functions can be expected to be implemented. For a minor misunderstanding, a few extra states may be assumed; for especially sensitive applications one can make very pessimistic assumptions about  $m$  at the cost of an exponentially larger test set. Rather often, states correspond to variables of a program and the part of the program being executed; in such cases, evaluation of coverage of values of the appropriate variables during testing and white-box coverage metrics may be used to judge whether any extra states have been missed.

### 3.3. Generation of the test set

Under these conditions, Theorems 3.1 and 2.3 can be used 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,

- $t$  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  $A_Z$ .

The following example illustrates the construction of  $X_{m-n}$  for  $Z$  as in Example 2.1 and  $m - n = 1$ .

**Example 3.2**  $S = \{\epsilon, \phi_1, \phi_1\phi_2\}$ ,  $P = S \cup S\Phi = \{\epsilon, \phi_1, \phi_2, \phi_3, \phi_1\phi_1, \phi_1\phi_2, \phi_1\phi_3, \phi_1\phi_2\phi_1, \phi_1\phi_2\phi_2, \phi_1\phi_2\phi_3\}$ ,  $W = \{\phi_1, \phi_2\}$ ,  $Y_1 = P(\Phi \cup \{\epsilon\})(W \cup \{\epsilon\})$ ,  $X_1 = t(Y_1)$ .

### 3.4. Test set size and length

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?

According to [Cho78],  $\text{card}(Y_{m-n}) \leq n^2 k^{m-n+1}$  and  $\text{length}(Y_{m-n}) \leq n^2 m k^{m-n+1}$ , 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  $\text{length}(X_{m-n}) \leq n^2 m k^{m-n+1}$ . In particular, for  $m = n$ , the respective bounds are  $n^2 k$  and  $n^3 k$ . Note that these bounds refer to the worst case. In an average case, the size of the test set is much lower.

It is worth noting that  $t(Y_{m-n})$  will, at most, yield the same number of test sequences of the same length. However in reality one might expect a substantial reduction in both of them. For instance, if some functions  $\phi_a, \phi_b$  are using the same input  $a$  to attempt them, identical sequences of inputs are obtained from different sequences of functions  $\phi_a\phi_b\phi_b, \phi_b\phi_a\phi_b$ .

An implementation has to feature a reliable reset. This condition is unchanged from the finite-state machine case. Reset after every sequence is not included in the calculations of a test set length. In practice, the time taken to reset a system can be substantially different time to the time taken to execute a transition; as a consequence, it does not seem reasonable to directly incorporate reset time in calculations of a test set length. Instead, both the total length of a test set and the number of test sequences are provided.

## 4. Industrial applications

The X-machine test method was applied in an industrial context (within a ‘Company’) over a period of 6 months. The devices to be tested are not public domain and the details are protected by a commercial non-disclosure agreement. However it is possible to summarise some of the results. The method was applied to designs being currently developed by the company. The test sets produced by this method were compared to the in-house test sets in the following way. Each design, represented by its VHDL source code, had a wide variety of mutation faults injected into it. The two test sets were then run on the mutated code and the results analysed. This was done for two designs that were being finalised. The X-machine test generation process was straightforward, taking less than a week per design – comparable with the in-house approach. The results were then analysed by generating a mutation factor score, which essentially measures the success of the two test methods in locating known faults in the VHDL code. Test execution has been performed on a simulator both for Company and X-machine testing. The Mutation score is shown as a percentage and is given by the following formula score  $S = \frac{D-Ed}{M-Em}$  where  $D$  is the number of dead mutants,  $Ed$  is the number of equivalent mutants that are also dead,  $M$  is the total number of mutants and  $Em$  is the number of equivalent mutants.

The main problem with both designs is that it is difficult to ensure that the  $\phi$  functions are error free. This is due to the fact that there is no clear distinction between control (represented by the transition diagram) and data transformations (performed by functions) and hence functions cannot be tested separately from testing of the state-transition diagram. The work described later in Sect. 5 aims to solve this problem, however it was completed after the described case studies took place.

Observability and state identification are easy to achieve since a simulator makes it possible to observe values of internal variables of a design. As a consequence,  $W$  set is not needed because states can be directly observed rather than checked by running sequences of transitions. In addition, it is not necessary to modify functions to satisfy input-completeness since (assuming that functions are correctly implemented) a device can be driven through sequences of functions to reach a desired state and memory value. Such sequences will not necessarily be those from a state cover set since longer sequences may be needed to set the memory value to the desired value.

### 4.1. Results: Design1

The first design, Design1, is a modern, real world example of a device that provides a challenge when it comes to testing. This case study compares the X-Machine test set with the test set developed by the Company that has been used to verify Design1 for commercial use. The Company test set aims at white-box coverage of VHDL constructs, supplemented by boundary-value testing. Even with design documentation, an X-Machine model needs to be reverse engineered from the documentation and design. Since the Company design process includes the design of a state machine to model the behaviour of the device, this made constructing the X-Machine model much easier; the design of the X-Machine is much closer to what would be expected if a device had been created from scratch using the X-Machine model. The model contains 8 states and 20 transitions, corresponding to 520 lines of VHDL code (30% are comments).

Table 1 shows how mutation operators can be used for the Design1 and coverage results. Table 2 gives coverage results for both the XM test sets and the Company test sets; it contains two groups of three columns, corresponding to the X-machine and the company test sets. For each of these groups, the third column is the total number of elements to be covered, the second one is the number of them which were executed by a test and the first column shows the percentage of elements which were covered (i.e.  $\text{executed}/\text{total} * 100\%$ ).

The mutation scores for the X-Machine test set are higher than the mutation scores for the Company test set, indicating that the X-machine method found more of the injected faults than the company’s method. Conversely, all the faults missed by the X-Machine test set were also missed by the Company test set. Of the four faults missed by the X-Machine test set, two were equivalent, leaving two faults that were actually missed. The Company test set left eight live mutants, most of which were faults that changed the initialisation behaviour of the device. This seemed to be an oversight in Company test methods until an engineer pointed out that initialisation tests are carried out manually after a design is completed and are not part of a test suite. In the opinion of the authors of this paper, if something can be tested automatically, it should be.

In this example, the higher the statement and branch coverage the more faults were found using the IF Branch and CASE Label mutation operators. Condition coverage does not seem to correlate, the higher code coverage of the Company test set results in the Company test set finding fewer of the faults introduced by the IF condition mutation operator. Again, this shows how the condition can be a misleading measure of the quality of a test set.

**Table 1.** Use of mutation operators for Design1 and the coverage results

Mutation operator	Mutations	Mutation score,%	
		X-Machine	The company
IF branch mutation	42	100	92
CASE label mutation	25	100	95
IF condition mutation	34	91	82

**Table 2.** Design1 coverage results

Coverage	X-Machine	X-Machine	X-Machine	The company	The company	The Company
	%	Executed	Total	%	Executed	Total
Statement	100	91	91	100	91	91
Branch	100	83	83	100	83	83
Basic sub-condition	100	46	46	100	46	46
Focused expression	91	42	46	100	46	46
Multiple sub-condition	47	41	88	53	47	88

From the measures taken in this case study, the two test sets seem to be similar in quality. The major difference between them is that the X-Machine test set takes approximately half the time to execute and requires a simpler test environment to execute test sequences. The design of the X-Machine compliant models took about 3 days and the X-Machine test set was generated automatically. The translation to the final test environment took a matter of hours to design. In contrast, the Company test set took one of their designers approximately 5–7 days to design. An experienced designer that knows how the device operates may be able to construct an X-Machine model and a test set much faster than an inexperienced hardware designer who is unfamiliar with the device in question (as was the case in these tests).

## 4.2. Results: Design2

Design2 is a relatively new device with a reasonably large state space that makes the verification task non-trivial. The device comes complete with design documentation used by the engineers and its own test set. With all this information, an X-Machine model needs to be reverse engineered from the documentation and design. The Company design process includes the construction of a state transition diagram to model the behaviour of Design2; this greatly helps the construction of the X-Machine model as the state space and transition structure are already defined. This brings the whole design process a little closer to the ideal situation where, a designer would construct a specification as an X-Machine and use it to construct the design. Design2 contains two state machines; two X-Machines are required to model its behaviour. These machines work in parallel and are connected together using a series of internal signals. In order to control the second-state machine, the internal signals had to be exposed; Design2 has been modified to achieve this. There is known work [PYvBD96] which describes how an implementation of a finite-state machine can be tested when it is surrounded by two correctly implemented FSMs, however for Design2, it is necessary to test the enclosing machine too. In total, for the two X-machines, there are 10 states and 31 transitions, which correspond to 717 lines of VHDL (34% are comments).

The Company test set consists of two parts: a structured test set and a random test set. The structured test set is designed in much the same way as for the Design1 and takes 1–2 h to execute. The random test set is generated by a C program that constructs test cases by randomly executing complete scenarios which Design2 can encounter during its operation, including scenarios where communication errors are encountered. The random part of the test set takes about 10–15 h to execute. Using the mutation metric on such a large test set was not practical.

Compared to the Company test set, the X-Machine test set is very low level. X-machine testing does not aim at running complete scenarios, but at testing of the transition diagram. This means that only parts of scenarios are run. As a result, test generation takes significantly less time than for the Company test set and executes much faster than the complete Company test set, taking only minutes as opposed to the overnight run the Company test set requires.

As the actual Company test set cannot be used, a portion of the random test set was generated to compare again the X-Machine test set. Two runs are performed. The first run is a randomly generated test set that takes the same amount of time to execute as the X-Machine test set. The second run is a randomly generated test set that is ten times longer than the first run. Although the X-Machine test set coverage results are encouraging at over 90%, the Company coverage results are well below this level. As this is not the complete test set, this is to be expected. However, the key issue here is not that the company's tests are incomplete but that for the same level of

**Table 3.** Design2 coverage results

Coverage	Short random test			Long random test			X-machine test		
	%	Executed	Total	%	Executed	Total	%	Executed	Total
Statement	85	127	149	87	131	149	97	144	149
Branch	81	142	175	86	150	175	95	166	175
Basic sub-condition	84	153	182	89	161	183	94	171	182
Focused expression	68	117	172	77	133	172	83	142	172
Multiple sub-condition	19	140	748	23	171	748	26	192	748

**Table 4.** Mutation scores for Design2

Mutation operator	Mutations	X-Machine, (%)	The Company, (%)
IF branch	49	86	56
CASE label	14	82	36
IF condition	38	84	68

‘effort’ (as defined by the time taken to execute the tests), the X-machine test set achieves better testing outcomes. The figures are shown in more detail in Tables 3 and 4.

In the case of the X-Machine test set, one iteration takes approximately 1 min. The Company test set takes longer to execute and so takes about 2 min to execute one complete iteration. In total, 101 iterations were performed resulting in 27 equivalent mutants and 74 unique mutants. Table 3 contains the overall results from the mutation testing. The first three columns of this table correspond to the first run of the random test, the second three – to the second run and the last three – to the X-machine test.

In this case study, the X-Machine test set performs significantly better than the Company random test generation method. Coverage results for the X-Machine test set are encouraging. Statement and Branch coverage are all at least 95%. The basic sub-condition coverage is 94%. This is as expected since the missing parts are mostly related to the computations performed by functions. The Company test set is much poorer in coverage performance with only statement coverage reaching 85%. The rest of the values fall far short of what is expected from a useful test set. This is important, because the X-machine testing method used in this case study did not aim to test functions or communication protocols utilised in Design2; despite this, it has achieved a rather better coverage than the Company test set. Increasing the duration of testing of Design2 by a factor of 10 only increases the coverage, on average, by 10%.

Similarly to experiments with Design1, in the case of Design2, the X-machine method found more of the injected faults than the company’s method and all the faults missed by the X-Machine test set were also missed by the Company test set. Table 4 shows a break down of the mutation scores for each mutation operator. There is little anomalous behaviour in these results and everything is as predicted. X-Machine mutation scores for all three mutation operators are high. The Company test set shows a very low score (36%) for case label mutations; X-machine testing scores 82%. Increasing the size of the random test set had no effect on the mutation scores for the Company test set.

### 4.3. Summary

The experiments with both designs produced consistently better results than Company tests in (almost) every case that was considered. X-machine testing seems also useful for testing of higher-level design properties, for example, capturing the kind of errors that designers routinely introduce such as the synchronisation of parts of a design. The testing method was also applied to a number of other microprocessors and other hardware designs that are in the public domain, together with standard test sets. Similar results were achieved; [Van02] describes them together with background information.

As described in this section, testing has primarily focused on testing with the goal of finding design defects. A different kind of testing is performed during manufacturing of semiconductors in order to detect manufacturing defects. Such a testing has a rather specific focus and it turns out that X-machine testing does not perform well in this domain.

A number of tools were developed to automate test generation (using Prolog), test application, mutation generation and the analysis. These were specific to the context described, see [Van02] for details.

As mentioned previously, X-Machine models for both designs were constructed from state machine diagrams and worded descriptions of functions, provided by the Company. Company engineers were not actually far from constructing X-Machine models themselves; to do this they simply needed (1) to take into account testing compatibility and design for test conditions (Definitions 3.3–3.5) when designing X-Machines and (2) describe

functions more precisely, such as using pseudo code. This implies that the approach can actually be used by the Company engineers without using a PhD student or an external consultant to build X-machine models and perform testing.

## 5. Complete DSXM testing

The complete DSXM testing method [Ipa04] generalises the DSXM integration testing method by making (almost) no assumptions 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.

### 5.1. Theoretical basis

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. Such a potentially incomplete specification describes the core behaviour of a system and an implementer is allowed to include arbitrary behaviours to elements of  $\Phi$  in an implementation, as long as the implementation remains deterministic. Additional functions can also be added, but they must be distinguishable from those in a specification. Clearly, if a specification is completely defined, then testing reduces to equivalence testing.

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 ( $\Sigma$ ) and output alphabet ( $\Gamma$ ) as  $Z$ . Then a finite set  $X \subseteq \Sigma^*$  is called a *weak test set* of  $Z$  w.r.t.  $C$  if  $\forall Z' \in C, (f_Z|X = f_{Z'}|X \implies f_Z \subseteq f_{Z'})$ .

Obviously, any test set of  $Z$  w.r.t.  $C$  (Definition 3.1) is also a weak test set of  $Z$  w.r.t.  $C$ .

The output-distinguishability condition has to be updated for the situation where a specification and an 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  $\Phi$  and  $\Phi'$ , respectively. Then  $\Phi$  is called *output-distinguishable w.r.t.  $\Phi'$*  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  $\phi$  in  $\Phi$ , we must be able to identify a corresponding function  $\phi' = c(\phi)$  in  $\Phi'$  by examining outputs. Note that (1) if  $\Phi = \Phi'$  then  $c$  is the identity function and (2)  $\Phi$  is output-distinguishable w.r.t. itself iff  $\Phi$  is output-distinguishable.

Unlike the assumption that functions have been tested in advance, it is only assumed in this section that all of  $\Phi'$  are output-distinguishable and that an implementation can be modelled as a DSXM (the latter implies that any two distinct processing functions that label arcs emerging from the same state have disjoint domains, restricting the additional behaviour which can be added to functions by an implementer).

In the definition of a weak test set (Definition 5.1), the condition  $f_Z|X = f_{Z'}|X$  is a sufficient, but not a necessary condition. Indeed, in a situation where a sequence of test inputs  $s\sigma$  is used such that  $s\sigma \notin \text{dom}(f_Z)$  but  $s \in \text{dom}(f_Z)$ , no output will be produced by a specification  $Z$  in response to  $\sigma$ ; at the same time, a compliant implementation may produce an output from either of

- $c(\phi) \setminus \phi$ , for some  $\phi$  (since in general  $\phi \subset c(\phi)$  may hold)
- any function from  $\Phi' \setminus \text{Im}(c)$ . This is possible if there are functions in  $Z'$  which are not an implementation of any function of a specification  $Z$ .

When  $X \subseteq \text{dom}(f_Z)$  (in particular when  $Z$  is completely defined), the condition  $f_Z|X = f_{Z'}|X$  is both necessary and sufficient. 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.  $C$ .

Further,  $A_{Z'}^c$  is used to denote the FA  $(\Phi', Q', F'_c, q'_0)$ , 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  $L_{A_{Z'}^c} \subseteq L_{A_{Z'}}$ .

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

$$\phi_1 \cdots \phi_n \in L_{A_{Z'}^{-c}} \iff c(\phi_1) \cdots c(\phi_n) \in L_{A_{Z'}^c} \iff c(\phi_1) \cdots c(\phi_n) \in L_{A_{Z'}}.$$

Since the method does not assume that 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 processing functions. This latter component will be called a *function test set*. The concepts necessary for its construction 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  $\phi$  w.r.t.  $\Phi'$*  if the following holds:  $\forall \phi' \in \Phi', (\forall \sigma \in \Sigma_m^\phi, (\pi_1(\phi(m, \sigma)) = \pi_1(\phi'(m, \sigma))) \implies \phi \subseteq \phi')$ .

A set  $\Sigma_m^\phi \subseteq \Sigma$  is called an  *$m$  test set of  $\phi$  w.r.t.  $\Phi'$*  if the following holds:  $\forall \phi' \in \Phi', ((\forall \sigma \in \Sigma_m^\phi, (\pi_1(\phi(m, \sigma)) = \pi_1(\phi'(m, \sigma)))) \implies \phi = \phi')$ .

That is, a (weak)  $m$  test set of  $\phi$  w.r.t.  $\Phi'$  is a finite set of inputs that tests  $\phi$  against any processing function in  $\Phi'$ .

**Definition 5.4** Let  $Z$  be a DSXM having type  $\Phi$ . Then a finite set  $V = \{v_1, \dots, v_k\} \subseteq \Phi^*$  is called a *function cover* of  $Z$  if  $\Phi$  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”  $\phi_i$ . 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 2.1,  $V = \{\epsilon, \phi_1, \phi_1\phi_2\}$  is a function cover of  $Z$ .

A function cover of  $Z$  exists if and only if for any proper subset  $\Phi_0$  of  $\Phi$ ,  $L_Z \setminus \Phi_0^* \neq \emptyset$ . This happens if  $A_Z$  is accessible and all processing functions in  $\Phi$  are actually used as labels in  $A_Z$  (i.e.  $\pi_2(\text{dom}(F)) = \Phi$ ), as 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  $X_f = \bigcup_{i=1}^k t(v_i)\Sigma_{m_i}^i$  is called a *function test set of  $Z$  w.r.t.  $\Phi'$*  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  $\phi_i$  w.r.t.  $\Phi'$ .

For simplicity, in the expression of  $X_f$   $t(v_i)$  was used instead of  $\{t(v_i)\}$ . A function test set of  $Z$  exists if a function cover of  $Z$  exists and  $\Phi$  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 a DSXM implementation against their specification.

We can assemble all these into Theorem 5.1 which is the theoretical basis for the complete DSXM testing method.

**Theorem 5.1** [Ipa04] Let  $Z$  be a DSXM having type  $\Phi$  input-complete and  $C$  a set of DSXMs weak testing compatible with  $Z$  having type  $\Phi'$  such that  $\Phi$  is output-distinguishable w.r.t.  $\Phi'$ . 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.  $\Phi'$ , then  $X' = t(Y) \cup X_f$  is a weak test set of  $Z$  w.r.t.  $C$ .

If the specification  $Z$  is completely defined, then  $f_Z$  is a total function, so  $X'$  establishes functional equivalence of specification and implementation rather than functional inclusion.

**Corollary 5.1** [Ipa04] In the conditions of Theorem 5.1, if  $Z$  is completely defined then  $X' = t(Y) \cup X_f$  is a test set of  $Z$  w.r.t.  $C$ .

It is also easy to see that if  $Z$  is completely defined then  $\phi_i = c(\phi_i)$ , so that weak test sets  $\Sigma_{m_i}^i$  are actually  $m_i$  test sets of  $\phi_i$  w.r.t.  $\Phi'$ .

## 5.2. Pre-requisites

The method works under the following assumptions:

1. The specification  $Z$  is a DSXM whose associated FA is minimal.
2. The type  $\Phi$  of the specification is input-complete and output-distinguishable.
3. The implementation can be modelled by a DSXM  $Z'$  such that  $\Phi$  is output-distinguishable w.r.t.  $\Phi'$ .
4. For any processing function  $\phi_i \in \Phi_i$  and memory value  $m_i \in M_i$ , a (weak)  $m_i$  test set of  $\phi_i$  w.r.t.  $\Phi'$  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 of states  $n$  of the DSXM specification.

The first two conditions refer to the DSXM specification and are identical to those for integration 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. This requirement is a consequence of usage of a specification-derived function cover. If an implementation uses different functions where a specification is using the same one, all these different functions have to be tested, but a function cover will only test one of them. For this reason, 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)  $m_i$  test set of a processing function  $\phi_i$  can be constructed using the same method or other functional testing approaches, depending on the complexity of  $\phi_i$ . 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.

### 5.3. Generation of the test set

Under these conditions, Theorems 5.1 and 2.3 can be used to generate a (weak) test sets for the DSXM specification. This 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  $A_Z$ ,
- $X_f = \cup_{i=1}^k t(v_i)\Sigma_{m_i}^i$  is a function test set of  $Z$  w.r.t.  $\Phi'$ , 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  $\phi_i$  w.r.t.  $\Phi'$ .

The following example illustrates the construction of  $X'_{m-n}$  for  $Z$  as in Example 2.1 and  $m - n = 1$ .

**Example 5.2** 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 3.2.

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

### 5.4. Test set size and length

In Sect. 3.4 it was shown that  $\text{card}(X_{m-n}) \leq n^2 k^{m-n+1}$  and  $\text{length}(X_{m-n}) \leq n^2 m k^{m-n+1}$ . 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  $\text{length}(\{v_1, \dots, v_k\}) \leq (n - 1)(k - 1)$ . 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  $\text{length}(X_f) \leq (n - 1)(k - 1)r + kr \leq nkr$ . Therefore  $\text{card}(X') \leq n^2 k^{m-n+1} + kr$  and  $\text{length}(X') \leq n^2 m k^{m-n+1} + nkr$ . In particular, for  $m = n$ , the respective bounds are  $n^2 k + kr$  and  $n^3 k + nkr$ .

## 6. Testing context

The testing method introduced in the preceding sections forces a developer to design software so as to make it possible to find a test input for every memory value. Instead, it could be possible only to use a subset of inputs and memory values for testing which could make attempting elements of  $\Phi$  easier in practice. For a relation  $\phi$ , inputs used for testing are denoted  $U_\phi$ ; memory values traversed during testing are assumed to be contained in  $V \subseteq M$ . Usage of a subset of inputs is only possible if

1. memory of an X-machine under test will stay within  $V$ .
2. it is possible to attempt every relation  $\phi \in \Phi$  using a subset of inputs,
3. outputs from these relations make it possible to distinguish between them.

The above three items are formalised using the idea of a *testing context*  $(V, U)$  following definitions from [HiH04, IpH00].

**Definition 6.1** Let  $(V, U)$  be such that  $V \subseteq M$  and  $U = \{U_\phi \subseteq \Sigma \mid \phi \in \Phi\}$ . Since the purpose of  $U_\phi$  is to contain inputs to attempt  $\phi$ ,  $\emptyset \notin U$ . The set  $\Phi$  is called *closed w.r.t*  $(V, U)$  if the following two conditions are satisfied,

- $m_0 \in V$ ;
- $\forall \phi \in \Phi, m \in V, \sigma \in U_\phi$ , if there is  $\gamma \in \Gamma, m' \in M$  such that  $(m, \sigma)\phi(\gamma, m')$ , then  $m' \in V$ .

**Definition 6.2** Let  $\Phi$  be closed w.r.t. some  $(V, U)$  where  $V \subseteq M$  and  $U = \{U_\phi \subseteq \Sigma \mid \phi \in \Phi\}$ . Then  $\Phi$  is *input-complete w.r.t*  $(V, U)$  if  $\forall \phi \in \Phi, m \in V, \exists \sigma \in U_\phi$  such that  $(m, \sigma) \in \text{dom}(\phi)$ .

**Definition 6.3** Let  $\Phi$  be closed w.r.t. some  $(V, U)$  where  $V \subseteq M$  and  $U = \{U_\phi \subseteq \Sigma \mid \phi \in \Phi\}$ . Then  $\Phi$  is *output-distinguishable w.r.t*  $(V, U)$  if  $\forall \phi_1, \phi_2 \in \Phi, m \in V, \sigma \in (U_{\phi_1} \cap \text{dom}(\phi_2)) \cup (U_{\phi_2} \cap \text{dom}(\phi_1))$ ,

$$\exists \gamma \in \Gamma, m'_1, m'_2 \in V. \text{ if } ((m, \sigma)\phi_1(\gamma, m'_1) \text{ and } (m, \sigma)\phi_2(\gamma, m'_2)), \text{ then } \phi_1 = \phi_2.$$

For  $V = M$  and a deterministic X-machine, the above two definitions reduce to input-completeness and output-distinguishability; the set  $\Phi$  of any X-machine is closed w.r.t  $(M, \Sigma)$  by definition of an X-machine. For these reasons, Definitions 6.2 and 6.3 are more general than Definitions 3.5 and 3.4.

Relations of a non-deterministic implementation can produce a number of different memory values as an outcome, but a tester may need to know the current memory in order to determine the appropriate input to attempt a relation. The observability condition (a part of output-distinguishability in [IpH00]) makes this possible.

**Definition 6.4** [HiH04]  $\Phi$  is *observable w.r.t.*  $(V, U)$  if  $\forall \phi \in \Phi, m \in V, \sigma \in U_\phi$ ,

$$\exists m_1, m_2 \in V. \text{ if } ((m, \sigma)\phi(\gamma, m_1) \text{ and } (m, \sigma)\phi(\gamma, m_2)), \text{ then } m_2 = m_1.$$

## 7. Testing of non-deterministic SXMs for equivalence

This section describes the extension of the X-machine testing method for testing of non-deterministic SXMs for equivalence [IpH00], assuming that elements of  $\Phi$  have been tested in advance and are equivalent to those of a specification.

The definition of a DSXM (Definition 2.16) requires determinism of an associated automaton, non-intersection of domains of functions from every state and elements of  $\Phi$  to be functions. As shown in Theorem 2.2, the equivalence of associated automata requires the equivalence of languages they accept; it is known [Coh96] that for every non-deterministic automaton (including the one with multiple initial states) it is possible to build a deterministic one, computing the same language. For this reason, it is possible to restrict the consideration to SXMs with a deterministic associated automaton. In a similar way to Sect. 3, testing is based on sequences of relations from  $\Phi$  being computed from an associated automaton using the W-method, followed by test data generation using a test function. Definitions 6.1–6.4 related to a testing context  $(V, U)$  from the previous section have to be satisfied in order for the test data to be computed. The main difference between testing DSXM and non-deterministic SXM is that when supplied with a sequence of inputs,

1. each relation executed can produce different memory values and outputs (due to arcs being labelled with relations rather than functions) and
2. a non-deterministic SXM can take various different sequences of relations (due to domain non-determinism).

For example, in software, a choice of a path can depend on subtle aspects of timing when a sequence of inputs is applied (synchronous hardware is typically designed to avoid non-determinism). It is assumed in this section that a tester cannot control such decisions made by an implementation. For this reason, a tester can either (1) observe which decision has been made by an implementation and adapt testing accordingly or (2) attempt sequences of inputs corresponding to the same test sequence a number of times in the hope that an implementation will eventually take the path aimed for by the tester.

Let a test sequence contain elements  $\phi_1 \dots \phi_n$  and  $i < n$ . A relation  $\phi_i$  can produce any of a (possibly infinite) set of memory values  $V$ . After executing  $\phi_i$ , an input to attempt  $\phi_{i+1}$  could depend on the specific value of memory produced by  $\phi_i$ . For this reason, it is unrealistic to solve the first case of non-determinism by attempting the same sequence of relation numerous times. This problem can be avoided altogether by tracking memory values, which is possible due to the observability requirement (Definition 6.4). Consequently, one has to define a *test process* which generates inputs depending on previously received outputs from a particular implementation under test. It is defined below following a few definitions.

The second case of non-determinism above is handled by attempting the same sequence of relations multiple times, since the number of possible decisions (paths) an implementation can take in response to the same input sequence is finite. The *complete-testing* assumption [LuBP94] is that by attempting same sequence of relations sufficiently many times, it is possible to get an implementation to go through all possible paths which it may take in response to the corresponding sequences of inputs (determined using a test process). It is assumed that a tester can determine a number  $c$  of attempts; sequences of inputs corresponding to a sequence of relations are attempted  $c$  times and if outputs from an implementation show that it did not execute the expected sequence of relations on any of the attempts, it is assumed that such a sequence of relations is not implemented.

**Definition 7.1** Two SXMs  $Z$  and  $Z'$  are called *nsxm testing compatible* if they have the same input ( $\Sigma$ ), output ( $\Gamma$ ) alphabets, memory sets  $M$ , initial memory values  $m_0$  and sets of relations  $\Phi$ .

**Definition 7.2** Let  $Z$  be a SXM and  $C$  – a set of SXMs which are nsxm testing compatible with  $Z$  with a known upper bound on the number of states in an associate automaton of  $Z' \in C$ . A set  $Y \subseteq \Sigma^*$  is called a *nsxm test set* if the following is satisfied:

$$\forall Z' \in C (\forall y \in Y, \text{ if there is a } g \in \Gamma^* \text{ such that } y \| f_Z \| g = y \| f_{Z'} \| g \text{ then } f_{Z'} = f_Z).$$

The definition of a test function (Definition 3.6) has to be extended here to cope with elements of  $\Phi$  being relations.

**Definition 7.3** Let  $Z$  be a SXM satisfying conditions of Sect. 6. Then a function  $t_n : \Phi^* \rightarrow \Sigma^*$  is called a *test function* of  $Z$  if the following hold:

- $t_n(\epsilon) = \epsilon$ ;
- for  $\phi_1, \dots, \phi_n \in \Phi$ , with  $n > 0$ ,  $t_n(\phi_1 \dots \phi_n) = \sigma_1 \dots \sigma_k$ , 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  $\sigma_i \in U_{\phi_i}$ , where  $k$  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  $\phi_1 \dots \phi_{i+1} \notin L_{A_Z})$ .

Given a sequence of relations  $p$  and an implementation  $Z'$ , a test process  $tp(Z')$  will attempt to follow  $p$ , at each stage choosing an input depending on the current value of memory. When an implementation executes a function  $\phi'$ , a tester will be able to tell  $\phi$  it corresponds to (from output-distinguishability) and the new memory value from the previous memory value, specification of  $\phi$  and observability (Definition 6.4). The previous value is either the initial one  $m_0$  or the one known from a sequence of functions taken by the implementation before  $\phi'$ .

**Definition 7.4** Let  $Z$  and  $Z'$  be nsxm-testing compatible SXMs (Definition 7.1) and  $Z$  is completely defined and satisfies conditions 6.1–6.4. Then a function  $tp(Z') : \Phi^* \rightarrow \Sigma^*$  is called a *test process* of  $Z$  and  $Z'$  and  $tp(Z')_{io} : \Phi^* \rightarrow \Sigma^*$  is called an *io-process* if the following hold:

- $tp(Z')(\epsilon) = \epsilon$ ;
- for  $\phi_1, \dots, \phi_n \in \Phi$ , with  $n > 0$ ,  $tp(Z')(\phi_1 \dots \phi_n) = tp(Z')_{io}(\phi_1 \dots \phi_k)$ , where  $k$  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  $\phi_1 \dots \phi_{i+1} \notin L_{A_Z})$ .

The function  $tp(Z')_{io}(\phi_1 \dots \phi_k)$  is determined by induction on the sequence of functions. The base case  $tp(Z')_{io}(\epsilon) = \epsilon$  and the implementation is expected to produce an empty output and memory value  $m_0$ . For some  $0 \leq j < k$  two cases are considered for the determination of  $tp(Z')_{io}(\phi_1 \dots \phi_{j+1})$ ,

- When an implementation followed the path  $\phi_1 \dots \phi_j$  and produced the expected outputs. Here there are  $m' \in M$ ,  $s = tp(Z')_{io}(\phi_1 \dots \phi_j)$  and  $g \in \Gamma^*$  produced by an implementation such that  $(m_0, s) \parallel \phi_1 \dots \phi_k \parallel (g, m')$  and  $(m_0, s) \parallel f_{Z'} \parallel (g, m')$ ;  $m'$  is the value of memory an implementation is expected to reach. In this case, the next input  $\sigma \in U_{\phi_{j+1}}$  is such that  $(\sigma, m') \in \text{dom}(\phi_{j+1})$  and  $tp(Z')_{io}(\phi_1 \dots \phi_{j+1}) = s\sigma$ .
- When an implementation has followed the path up to and including element  $e < j$  of it, but afterwards either (1) executed a different relation than  $\phi_{e+1}$  or (2) ignored an input because there was no function implemented in it from the state it was in for the particular value of memory. Here a tester has to stop the test sequence after an input which attempted  $\phi_{e+1}$ . Formally, for  $s = tp(Z')_{io}(\phi_1 \dots \phi_{e+1})$ ,  $e < j$  there are no  $m' \in M$  and  $g : \Gamma^*$  satisfying both  $(m_0, s) f_{Z'} \parallel (g, m')$  and  $(m_0, s) \parallel \phi_1 \dots \phi_{e+1} \parallel (g, m')$ , but there are  $m'_1 \in M$ ,  $s_1 \in \Sigma^*$  and  $g_1 \in \Gamma^*$  such that  $s_1 = tp(Z')_{io}(\phi_1 \dots \phi_e)$ ,  $(m_0, s_1) \parallel \phi_1 \dots \phi_e \parallel (g_1, m'_1)$  and  $(m_0, s_1) \parallel f_{Z'} \parallel (g_1, m'_1)$ . In this case,  $tp(Z')_{io}(\phi_1 \dots \phi_{j+1}) = s$ .

## 7.1. Pre-requisites

1. An implementation  $Z'$  can be modelled by a SXM with the same input, output alphabets, memory sets and initial memory values as a specification  $Z$ . Additionally,  $Z'$  it has to have the same set of relations  $\Phi$  of  $Z$  (same as for DSXM testing in Sect. 3, but here elements of  $\Phi$  can be relations which are not necessarily functions). Both an implementation and a specification can be non-deterministic.
2. The complete-testing assumption is satisfied.
3. An associate automaton of a specification  $A_Z$  is deterministic and minimal. This requirement is the same as for DSXM testing.
4. A specification  $Z$  is a completely defined SXM; an implementation does not have to be complete because for a test set  $Y$ ,  $\text{pref}(Y)$  is used (refer to the discussion after Theorem 2.3 for details).
5. There is a known upper bound on the number of states in an implementation.
6. Conditions of Sect. 6 (conceptually similar to design for test conditions, introduced for DSXM testing) are satisfied. This means that there is a testing context  $(V, U)$  such that  $\Phi$  is closed, input-complete, output-distinguishable and observable w.r.t  $(V, U)$ . Above, the testing context was introduced for relations  $\phi : M \times \Sigma \longleftrightarrow \Gamma \times M$ ; [IpH00] shows that a test method will find all faults for generalised SXMs, where  $\phi : M \times \Sigma \longleftrightarrow \Gamma^* \times M$  and the notion of a testing context is modified accordingly.

## 7.2. Generation of the test set

Test generation consists of two parts:

1. Generation of a set  $Y_{m-n}$  following the W-method, where  $m$  is the upper bound on the number of states in an implementation and  $n$  is the number of states in a specification.
2. Computation of  $t_n(Y_{m-n})$  using a test function (Definition 7.3).

**Theorem 7.1** Under the conditions listed above, the set  $t_n(Y_{m-n})$  is a nsxm test set (Definition 7.2).

## 7.3. Test set size and length

The size of a test set for testing of non-deterministic SXMS for equivalence is directly related to that for DSXM testing, with the only difference being that every sequence of relations has to attempted a number of times (perhaps with different inputs). If the maximal number of attempts is  $c$ ,  $y = \text{card}(Y_{m-n})$  and  $l = \text{length}(Y_{m-n})$  are derived in Sect. 3.4, then the number of test sequences is  $cy$  and their total length is  $cl$ .

## 8. Testing of non-deterministic SXMs for conformance

This section describes the extension of the X-machine testing method for testing of non-deterministic SXMs for conformance [HiH04] (this paper supersedes [HiH00]). This type of testing is important in situations where a specification gives an implementer a choice of possible behaviours to implement. Such a specification is generally non-deterministic and conformance testing aims to demonstrate  $f'_Z \subseteq f_Z$  (under the condition that both are completely defined) rather than  $f_Z \subseteq f'_Z$ , as has been considered in Sect. 5. Completeness restricts the number of elements an implementer can remove from  $f_Z$  without making it incomplete; it also implies that no extra functions can be added in an implementation. If  $f_Z$  is a function, there is only one allowed output for every input and hence conformance testing reduces to equivalence testing. It is important to mention that the described testing method does not test functions; it is assumed that they are tested for conformance separately.

Sections 5 and 8 assume two different models of how software or hardware can be built. The former considers the case where one starts with a partially-specified model (describing a core behaviour) and completes it during implementation. Hence testing is for inclusion  $f_Z \subseteq f_{Z'}$ ; it is possible that  $\text{dom}(f_Z) \subseteq \text{dom}(f_{Z'})$ . The current section considers an alternative approach where one starts with a non-deterministic specification describing all allowed behaviours, from which an implementer can choose any. For this reason, testing is for  $f_{Z'} \subseteq f_Z$ ; both  $Z$  and  $Z'$  have to be completely defined. Following from the discussion of the role of the condition  $f_Z|X = f_{Z'}|X$  of Definition 5.1, in the situation where both  $Z$  and  $Z'$  are completely defined and  $Z$  is a DSXM, both testing methods test for equivalence of the behaviour of  $Z$  and  $Z'$ .

For a specified function  $\phi$  any implemented function does not have to be equivalent to some specified one, but has to correspond to one. For some implementation  $\phi'$  of  $\phi$ , Sect. 5 requires  $\phi \subseteq \phi'$ ; for conformance (this section), the reverse inclusion is needed,

**Definition 8.1** For  $\phi \in \Phi$  and  $\phi' \in \Phi'$ ,  $\phi'$  conforms to  $\phi$  (written as  $\phi' \leq \phi$ ) if  $\phi' \subseteq \phi$  and  $\text{dom}(\phi') = \text{dom}(\phi)$ . Additionally,  $\Phi' \leq \Phi$  if  $\forall \phi' \in \Phi'$ , there is  $\phi \in \Phi$  such that  $\phi' \leq \phi$ .

This definition can be extended pointwise to sequences of functions, where it is denoted by  $\leq^*$ .

**Definition 8.2** Two SXMs  $Z, Z'$  are called *ct-testing compatible* if

1. They are completely defined;
2.  $Z'$  has the same input, output alphabets, memory sets and initial memory values as  $Z$ ;
3.  $Z'$  computes a (total) function  $f_{Z'}$  and has a deterministic  $A_{Z'}$  (it does not have to be a DSXM);
4.  $\Phi' \leq \Phi$ .

Definitions 6.1–6.4 related to a testing context  $(V, U)$  apply here too. Moreover, one can use the definition of an adaptive test process (Definition 7.4) from the previous section because the requirements of ct-test compatibility imply nsxm-test compatibility (Definition 7.1) and complete-definedness of a specification.

**Definition 8.3** Let  $Z$  be a completely defined SXM and  $C$  be a set of ct-testing compatible SXMs for which there is a known upper bound on the number of elements in their associate automata. Then for  $Z' \in C$ ,  $\text{ctp}(Z')$  is called a *conformance test process* of  $Z$  w.r.t.  $Z'$  if  $(\forall x \in \text{ctp}(Z'), (f_{Z'}(x) \in f_Z(x)) \implies (\forall x \in \Sigma^*, (f_{Z'}(x) \in f_Z(x)))$ .

Definitions 6.1–6.4 apply to a specification; it immediately follows from  $\Phi' \leq \Phi$  that all but output-distinguishability apply to an implementation. It is possible for multiple implemented relations to conform to the same one in a specification; in this case, these implemented relations will not be output-distinguishable. This is allowed because (1) for any implemented relation, it is enough to identify a specified relation it conforms to; (2) relations are not tested using the function cover approach, hence one does not have to test all the different implementations of the same relation. Section 5 had to introduce output-distinguishability of  $\Phi$  w.r.t  $\Phi'$ , because extra functionality added by an implementer can make elements of  $\Phi'$  not output-distinguishable; here elements can only be removed from relations. Testing-completeness cannot be violated here because conformance requires domains of implemented relations to be equal to those of conforming relations of a specification.

If a specification satisfies the above conditions, the following two lemmas show that a tester can attempt an arbitrary sequence of relations and check if a corresponding path (consisting of conforming relations) exists from the current state in an implementation.

**Lemma 8.1** [HiH04] Let a specification  $Z$  satisfy the conditions of Sect. 6. For any  $p \in \Phi^*$ , there is a sequence  $s \in \Sigma^*$  such that there exists  $m \in M, g \in \Gamma^*$  satisfying  $(m_0, s) \parallel_p \|(g, m)$ .

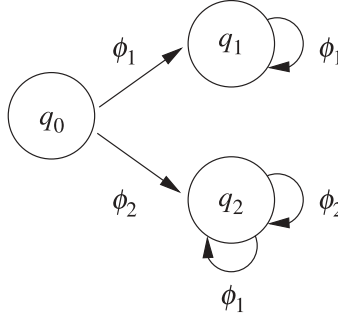


Fig. 2. The associated FA of a non-deterministic SXM

**Lemma 8.2** [HiH04] Let a specification  $Z$  satisfy the conditions of Sect. 6. For any  $p' \in \Phi'^*$  in a ct-testing compatible implementation  $Z'$ , there is one and only one  $p \in \Phi^*$ , satisfying  $p' \leq^* p$ .

The last lemma makes it possible to define a function  $-c$ , mapping  $\phi' \in \Phi'$  to  $\phi \in \Phi$ . This function can be considered an inverse of  $c$  introduced in Definition 5.2; the main difference is that here  $-c$  is a function and  $c$  is not necessarily a function, while in Sect. 5 both  $c$  and  $-c$  are injective functions. In a similar way to Sect. 5 it is possible to build an automaton  $A_Z^{-c} = (\Phi, Q', F'_{-c}, q'_0)$ .

The theorem below shows that testing for conformance can be reduced to testing of associated automata, but such a testing is for language inclusion rather than equivalence.

**Theorem 8.1** Let a specification  $Z$  satisfy the conditions of Sect. 6 and an implementation  $Z'$  be ct-compatible with  $Z$ . In this case,  $f_{Z'} \leq f_Z$  iff  $L_{A_Z^{-c}} \subseteq L_{A_Z}$ .

In a similar way to the equivalence testing, it is possible to convert a (non-deterministic) associated automaton of a specification into a deterministic one. In testing of DSXM, an equivalent implementation cannot have less states than a specification; here a conforming implementation can have fewer states.

**Example 8.1** A SXM  $(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$  with  $\Sigma = \{a\}$ ,  $\Gamma = \{x\}$ ,  $Q = \{q_0, q_1, q_2\}$ ,  $M = \{0\}$ ,  $m_0 = 0$ ,  $\Phi = \{\phi_1, \phi_2\}$ . Let  $\phi_1$  and  $\phi_2$  be defined for all values of inputs and memory, and  $F$  be as shown in Fig. 2.

An implementation of the specification shown in Fig. 2 is allowed to have  $Q = \{q_0, q_1\}$  (with either of the two transitions from  $q_1$  implemented),  $\{q_0, q_2\}$  or just  $q_0$  with a transition labelled  $\phi_1$  looping in it.

If for some  $s \in \Sigma^*$ , there is a unique output defined by  $f_Z$ , then a conforming implementation has to contain it and an associated sequence of relations. Such sequences of relations which have to be implemented by any conforming implementation can be derived from a specification and used during testing both to deterministically enter states and to distinguish between states. Distinguishing between states is a substantially different problem to the one described so far, because not every pair of states can be distinguishable. For instance, in Example 8.1 (Fig. 2), any of  $\phi_1$  or  $\phi_2$  can be implemented from  $q_1$  (assuming  $q_1$  is itself implemented). Consequently, neither of them can be used to distinguish between  $q_0$  and  $q_1$ . In some finite-state machines, there are states that are indistinguishable in a specification, but are always distinguishable in a conforming deterministic implementation [Hie98]. This result is not extended to X-machines in [HiH04] and hence is not considered in this paper.

**Definition 8.4** Consider a testing context  $(V, U)$ . For a state  $q \in Q$ , a set  $LD_Z(q)$  is defined to consist of sequences  $\phi_1, \dots, \phi_n \in \Phi$  satisfying the property that for every  $m \in V$  and  $\sigma_1, \dots, \sigma_n \in \Sigma$ , where  $\sigma_i \in U_{\phi_i}$ , the following holds:

$$(m, \sigma_1 \dots \sigma_n) \notin \bigcup_{p_1 \in (L_{A_Z}(q) \setminus \{p\})} \text{dom}(p_1)$$

As described above, sequences in  $LD_Z(q_0)$  make it possible to reach certain states of a specification for certain. Such states are referred to as *deterministically reachable*, abbreviated d-reachable.

**Definition 8.5** A set  $V \subseteq LD_Z(q_0)$  is a *deterministic state cover* if no state of  $Z$  is reached by more than one sequence in  $V$ .  $V$  has to contain an empty sequence  $\epsilon$ . For a chosen  $V$ , a set of d-reachable states is denoted by  $S_V \subseteq Q$ .

The initial state is always deterministically reachable as there is assumed to be only one initial state and it is reached by an empty sequence.

**Definition 8.6** Two states  $s_i, s_j \in Q$  are distinguishable in  $A_Z$  if there exists  $p \in LD_Z(s_i)$  such that  $p \notin L_{A_Z}(s_j)$ ,

It is proven [HiH04] that distinguishable states can actually be distinguished in an implementation. With this, a  $W$  set can be chosen such that for every distinguishable pair of states, there will be a sequence in it to distinguish them; unlike testing of DSXM, not all states can be distinguished.

Conformance testing between finite-state machines has been studied by [PeYB96] which introduced the state-counting (SC) testing method; [HiH04] uses this method for testing of  $L_{A_{Z'}^c} \subseteq L_{A_Z}$ . In a similar way to testing of FA, an upper bound  $m$  on the number of states in  $L_{A_{Z'}^c}$  is assumed to be known. Testing of deterministic X-machines involves entering every state of an implementation, attempting all possible relations from them and checking if both a specification and an implementation have taken them. The problem described in this section does not permit this to be directly applied to an implementation, since it may have states which are not deterministically reachable (any of a number of paths can be implemented in an implementation) and a number of states will not necessarily be distinguishable. Given two known FA,  $A_Z$  and  $A_{Z'}^c$ , one can build a cross-product of their states, such that states  $(q_i, q'_i)$  of a cross-product correspond to pairs of states  $q_i, q'_i$  in the two FA. A transition  $(q_i, q'_i) \xrightarrow{a} (q_{i+1}, q'_{i+1})$  exists only if both transitions  $q_i \xrightarrow{a} q_{i+1}$  and  $q'_i \xrightarrow{a} q'_{i+1}$  exist. The result of such a construction corresponds to an intersection of languages of the two automata. If languages accepted by  $A_Z$  and  $A_{Z'}^c$  are different, there will be a transition from one of  $(q_i, q'_i)$  which only one of the two automata can follow. Here the aim is to check language inclusion between the two automata, hence the task is to find a state in  $(q_i, q'_i)$  and a relation which can be taken by  $A_{Z'}^c$ , but not by  $A_Z$ . For this reason, testing for language inclusion may be considered to correspond to testing of a product automaton in search for such transitions. This can be done by entering each of its states and trying every relation from it; since the aim is solely to find transitions which should not be implemented, there is no need to verify entered states. By adding transitions which can be taken by a specification but not an implementation  $(q_i, q'_i) \xrightarrow{a} (q_{i+1}, q'_i)$  and the other way around  $(q_i, q'_i) \xrightarrow{a} (FAIL)$ , the described testing problem reduces to testing of a product FA (denoted  $PFA$ ) in order to find if the  $FAIL$  state is reachable.

**Theorem 8.2** [HiH04] Let  $Z$  and  $Z'$  be ct-testing compatible SXMs and  $Z$  satisfies conditions of Sect. 6.  $L_{A_{Z'}} \subseteq L_{A_Z}$  if and only if the state  $FAIL$  of the PFA cannot be reached from the initial state of the PFA.

During testing  $A_{Z'}$  is not known, it is only known that  $Z'$  is ct-testing compatible to  $Z$ , satisfies conditions of Sect. 6 and has a known upper bound on the number of states in  $A_{Z'}^c$ . For this reason, the main problem is to derive test sequences  $\Upsilon$  to visit every state of a non-deterministic PFA. Using the bounds on the number of states in  $A_{Z'}^c$  and  $A_Z$  yields  $\Phi^{mn}$  which is generally a very large set. Although there are cases [LuPB94] when one has to attempt all of those sequences, most of the time this is not going to be the case. In such cases, sequences through the PFA will loop before reaching the length  $mn$ . For every sequence, if a length can be derived from a specification after which such a sequence is certain to loop, the part of the sequence after a loop can be discarded without an impact on the ability of the sequence to reach all states in a PFA. There are two approaches which can be used to find when a particular sequence will loop.

1. Some states could be distinguishable (Definition 8.6); let a set of pairwise distinguishable states be  $T$ . If a path  $p$  visits some states of  $T$ , a tester can use  $W$  after every prefix of  $p$  to distinguish between states of  $T$  visited along  $p$  in an implementation. If states  $q^j \in T$  were visited  $a^j$  times along  $p$  in a specification, they will be visited the same number of times in an implementation. If the total number of times  $\sum_j a^j$  states from  $T$  were encountered is above  $m$ , an implementation must have entered a loop.
2. There could be some deterministically reachable states and the corresponding states in an implementation can be reached by sequences from  $V$ ; for this reason, it seems reasonable to explore a PFA starting from these states. Moreover, if a sequence  $p$  from a state  $s_1 \in S_V$  enters another  $s_2 \in S_V$ , it does not make sense to extend  $p$  because it is known for certain how to reach  $s_2$  (and a sequence in  $V$  to enter  $s_2$  will not be longer than  $p$ ). It is possible to combine this idea with the above one, such as to attempt sequences from  $VW$  and then cover all paths starting from  $S_V$  states, such that states in  $T$  are met at most  $m - \text{card}(T \cap S_V)$  times. This follows from the observation that if there are  $T$  pairwise distinguishable states,  $VW$  ensures that those of  $T$  which are deterministically reachable are indeed reached; this leaves  $\text{card}(T \cap S_V)$  fewer states in  $A_{Z'}^c$  to explore. In other words, the bound of  $m$  on the number of times states from  $T$  were encountered along a path can be reduced by  $\text{card}(T \cap S_V)$ .

When a specification is deterministic, testing for conformance reduces to testing for equivalence, hence the described testing method is a generalisation of the DSXM testing method (Sect. 3).

### 8.1. Pre-requisites

The conditions under which all faults will be found by the described conformance testing method are listed below:

1. Both a specification  $Z$  and an implementation  $Z'$  are completely defined SXMs.
2.  $Z'$  has the same input, output alphabets, memory sets and initial memory value as  $Z$ .
3. Associated automata of both a specification  $A_Z$  and an implementation  $A_{Z'}$  are deterministic.
4. There is a known upper bound on the number of states in an implementation.
5.  $Z'$  computes a (total) relation  $f_{Z'}$  (it does not have to be a DSXM, because a composition of relations which are not functions can compute a function).
6.  $\Phi' \leq \Phi$ , i.e.  $\forall \phi' \in \Phi' (\exists \phi \in \Phi (dom(\phi') = dom(\phi) \text{ and } \phi' \subseteq \phi))$ .
7. Conditions of Sect. 6 are satisfied. This means that there is a testing context  $(V, U)$  such that  $\Phi$  is closed, input-complete, output-distinguishable and observable w.r.t  $(V, U)$ .

### 8.2. Generation of the test set

The intuitive rules for the construction of  $\Upsilon$  introduced above to visit all states of a PFA can be formally written down as follows.

**Definition 8.7** Sets  $\Upsilon_v \in \Phi^*$  for all  $v \in S_V$  are such that

1.  $\forall p \in \Upsilon_v$ , there is some set of pairwise distinguishable states  $T$  of  $A_Z$  and the number of times states from  $T$  are encountered along a path  $p$  in  $A_Z$  is at least  $m - card(T \cap S_V)$ .
2. For all  $l \in L_{A_Z}$ , there is  $v \in S_V$  and a  $p \in \Upsilon_v$  such that either  $vp \in pref(l)$  or  $l \in pref(vp)$ .

The main result [HiH04] is that a test process (Definition 7.4) applied to the intuitively-described test suite mentioned above will find all faults,

**Theorem 8.3** Under the above mentioned conditions, let

$$E = \bigcup_{v \in S_V, p \in \Upsilon_v} pref(vp)(\Phi \cup W).$$

Then a test process  $tp(Z')(E)$  is a conformance test process  $ctp(Z')(E)$  of  $Z$  w.r.t.  $C$ .

### 8.3. Test set size and length

It is clear that the maximal length of test sequences to visit all states is  $mn$ . Apart from these observations, it is not easy to give a precise size of a test set, since everything depends on how many distinguishable states are traversed by paths in  $Z$ .

## 9. Testing of communicating SXMs for equivalence

This section describes how the X-machine testing method can be used for testing of deterministic communicating SXMs [IH02]. The model of communicating X-machines is different from the model described in [LuBP94] in that states are subdivided into two kinds, communicating and non-communicating; transitions from communicating states are only allowed to perform communications; those from non-communicating states are not allowed to communicate. All queues are effectively limited to the length of 1.

A communicating X-machine system (CSXMS) consists of a number of communicating X-machines  $P_i$  communicating with each other. Every communicating XSM (referred to as CSXM)  $P_i$  features two types of states, communicating  $Q_i^c$  and ordinary  $Q_i^n$  ones, sets of which do not intersect. Relations are also subdivided

into two non-intersecting sets, communicating relations  $\Phi_i^c$  and non-communicating ones  $\Phi_i^n$ ; the former can only label transitions from communicating states; non-communicating relations can only label transitions from non-communicating states. Communication channels are of length 1 and are arranged as a square matrix  $C$  with  $n \times n$  elements where  $n$  is the number of communicating SXMs. An element with an index  $ij$  stores a datum to be passed from machine  $i$  to machine  $j$ . An element can only be placed into a slot of  $C$  or retrieved from it by a communicating relation, of which there are two,  $mvo_{ij}$  and  $mvi_{ij}$ . The former puts an element into one of  $c_{ij}$ ,  $1 \leq j \leq n$ ; the latter retrieves one from  $c_{ji}$ . Restrictions on which elements of  $C$  can be read and written ensure that an element in a matrix is readable by only one CSXM and it is writable by only one (different) CSXM. Memory of CSXM is not accessible to the two communicating functions, which transfer data from a so-called *output port* of a CSXM to elements of  $C$  and place data retrieved from  $C$  in an *input port*. Both ports and memory are accessible to non-communicating relations. When no communication from CSXM  $i$  to  $j$  can occur, the appropriate element  $c_{ij}$  of the communication matrix has  $\theta$  in it. For instance, [IH02] prohibits any machine from communicating with itself. When a communication from  $i$  to  $j$  is possible, but the channel is empty,  $c_{ij} = \lambda$ .

The purpose of the separation of relations into communicating and non-communicating ones is to make it possible to synchronise multiple communicating SXM. An ordinary SXM requires an input for every transition, however one would like to make a machine wait for another one for an unknown period of time without consuming a number of inputs. There are two cases when a CSXM has to wait, (1) if it wishes to send data to another one but the channel between them is not empty and (2) when it would like to receive data but the channel is empty. In the model of CSXMS, a CSXM is allowed to remain in a communicating state and consume no input when it cannot take any transition; in contrast, when it is not in a communicating state it has to consume an input and take a transition. When a CSXM receives an input in a non-communicating state for which no transition can be taken, such a CSXM is assumed to halt and produce no output until it is reset and a new sequence of inputs is supplied to it. Hence CSXMs are completely defined in that upon receipt of an input which causes no transition to be taken from an ordinary state, they halt. In contrast, SXMs considered in Sects. 3 and 7 are assumed to ignore inputs which do not cause a transition. It is possible for the whole CSXMS to deadlock (such as when all X-machines are waiting to receive data from each other) or to halt (when all CSXMs halt). It is assumed that both of these cases can be detected by a tester (such as by means of timeouts as mentioned in [LuBP94]) and will correspond to an empty output being produced.

**Definition 9.1** [IH02] A communicating X-machine system (CSXMS) consists of a number of X-machines  $P_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, m_i^0, IN_i, OUT_i)$ , communicating with each other and satisfying the following properties:

- $Q_i = Q_i^c \cup Q_i^n$ ,  $Q_i^c \cap Q_i^n = \emptyset$ .
- $\Phi_i = \Phi_i^c \cup \Phi_i^n$ ,  $\Phi_i^c \cap \Phi_i^n = \emptyset$ .
- $dom(F_i) = (Q_i^c \times \Phi_i^c) \cup (Q_i^n \times \Phi_i^n)$ .
- Non-communicating relations  $\phi \in \Phi_i^n$  are such that  $((in_i, m_i, out_i, c), \sigma)\phi(\gamma, (in'_i, m'_i, out'_i, c))$ . Elements  $in_i \in IN_i$ ,  $out_i \in OUT_i$  are ports; since every two machines have a channel between them, it is assumed that  $\lambda \in IN_i$ ,  $\lambda \in OUT_i$ . The communication matrix  $c$  is unchanged by relations from  $\Phi_i^n$ .
- $mvo_{ij}((in_i, m_i, out_i, c), \epsilon) = (\epsilon, (in_i, m_i, \lambda, c'))$  where  $c'_{ij} = out_i$  and other elements of  $c$  are unchanged. No input is required for communicating transitions with function  $mvo_{ij}$  to fire. Memory  $m_i$  is unchanged by  $mvo_i$ .
- $mvi_{ij}((in_i, m_i, out_i, c), \epsilon) = (\epsilon, (\lambda, m_i, out_i, c'))$  where  $in_i = c'_{ij}$  and  $c'_{ij} = \lambda$  with other elements of  $c$  being unchanged. Memory  $m_i$  is unchanged by  $mvi_i$  and no input is consumed by it.
- $\Phi_i^c \subseteq \{mvi_{i1}, \dots, mvi_{in}, mvo_{i1}, \dots, mvo_{in}\}$ . It is assumed that if  $IN_i = \{\lambda\}$ ,  $mvi_{ij}$ ,  $1 \leq j \leq n$  are not a part of  $\Phi_i^c$  and similarly for  $OUT_i = \{\lambda\}$  and  $mvo_{ij}$ .
- A communicating stream X-machine system is a tuple  $S_n = (P_1 \dots P_n, C, c^0)$ . The initial value  $c^0$  of  $C$  has  $\theta$  on a diagonal (to stop machines communicating with themselves) and the rest of  $c^0_{ij}$  are  $\lambda$ . The behaviour of CSXMS is organised in steps, where in each step (1) each machine takes at most one transition and (2) at least one of the CSXMs takes a transition. The latter condition ensures that if all communicating SXMs are in their communicating states and preconditions of communicating transitions are satisfied, at least one communication has to take place.

Each of the X-machines in a CSXMS has its own input and output streams and since communications do not consume inputs or produce outputs, these streams may not be synchronised. If one is to consider a global state

for a particular CSXMS, this will be  $(z_1 \dots z_n, C)$ , where  $z_i = (m_i, q_i, in_i, out_i, s_i, g_i)$ . Here  $s_i, g_i$  are the input and output streams of the  $i$ th CSXM, respectively; when a SXM corresponding to  $z_i$  operates, inputs are being removed from a sequence  $s_i \in \Sigma_i^*$  and outputs are being appended to  $g_i \in \Gamma_i^*$ .

For sets  $D_i$ , let  $CROSS(D)$  mean  $(D_1 \cup \{\epsilon\}) \times (D_2 \cup \{\epsilon\}) \times \dots \times (D_n \cup \{\epsilon\}) \setminus \{(\epsilon, \dots, \epsilon)\}$ . In a similar way to relation  $f_Z$  (Definition 2.17), a relation  $f_Z^{cx}$  computed by a CSXMS can be defined as  $CROSS(\Sigma) \parallel f_Z^{cx} \parallel CROSS(\Gamma)$ . At every moment in time, CSXMSs which are not in their communicating states consume inputs, take transitions and append outputs to their output streams; SXMSs which are in their communicating states either perform communications (as per *mvo* and *mvi*) or do nothing. An empty input  $(\epsilon, \dots, \epsilon)$  is not included because it is assumed that after every input a CSXMS makes an arbitrary number of steps consisting of communications without consuming inputs.

A CSXMS introduced above can choose to take an infinite sequence of communicating transitions; in addition, communicating functions do not satisfy the design for test conditions of Sect. 6. For this reason, the following is proposed in [IH02].

**Definition 9.2** Given a CSXMS  $S_n$ , a testable CSXMS  $S_n^{Ta}$  can be constructed from it as follows:

1. Not every communicating function can be attempted from every state, for instance, because they require elements of the communicating matrix to be  $\lambda$  (for *mvo*-functions) and non- $\lambda$  (for *mvi*). It is assumed that unlike ordinary relations, forcing preconditions of communicating functions to be satisfied via design for test is not reasonable. For this reason, for *mvo* <sub>$i$</sub>  a tester needs a mechanism to find out when  $out_i \neq \lambda$  and  $c_{ij} = \lambda$ ; for *mvi* <sub>$i$</sub> ,  $in_i = \lambda$  and  $c_{ji} \neq \lambda$  are required. In order for conditions related to ports  $out_i$  and  $in_i$  to be satisfied, ordinary relations have to be modified to empty the input port and always put a non- $\lambda$  value in the output one. To satisfy the conditions associated with the communication matrix, new states are introduced, recording whether a particular channel is empty or not. There are  $n \times n$  of them, arranged in a matrix  $EM$  such that  $em_{ij} \in \{1, \lambda\}$ . An element  $em_{ij} = 1$  means that  $c_{ij} \neq \lambda$  and  $em_{ij} = \lambda$  means that  $c_{ij} = \lambda$ . With the described changes, function *mvo* <sub>$ij$</sub>  can only be taken from a state with  $em_{ij} = \lambda$  and sets  $em'_{ij} = 1$ ; *mvi* <sub>$ij$</sub>  may only label a transition from  $em_{ij} = 1$  and set  $em'_{ij} = \lambda$ .
2. The initial states of all CSXMSs have to be non-communicating,  $I_i \subseteq Q_i^n$ . This prevents CSXMSs from taking transitions before receiving any input and hence ensures a unique initial value of memory.
3. For any  $\phi_i^c \in \Phi_i$  and  $q_i^c \in Q_i^c$ , such that  $F_i(q_i^c, \phi_i^c) = q_i$ ,  $q_i \in Q_i^n$ . This means that any communicating function enters a non-communicating state. For a system with  $n$  CSXMS, this guarantees that the maximal number of relations they may execute without consuming any inputs is  $n$ . This ensures that the complete-testing assumption is satisfied by eliminating potentially infinite sequences of communicating transitions and thus avoids problems with testing of transitions which feedback [Hie97a].
4. Since communicating functions do not require an input to be attempted and produce no output, for testing they have to be replaced with functions which require an input to fire; additionally, they have to produce an output, making it possible to identify such functions. A function  $\phi \in \Phi_i^c$  such that  $\phi((in_i, m_i, out_i, c), \epsilon) = (\epsilon, (in'_i, m_i, out'_i, c'))$  is replaced with

$$\phi^{Ta}((in_i, m_i, out_i, c), a) = (o_i(\phi), (in'_i, m_i, out'_i, c')).$$

Here  $a$  is a new input  $a \notin \cup_{1 \leq i \leq n} \Sigma_i$ , introduced to attempt communicating functions and  $o_i(\phi)$  is an output uniquely identifying  $\phi$  among all relations of all  $P_i$ ,  $1 \leq i \leq n$ . Note that all of  $\phi^{Ta}$  are attempted using the same input. It is further assumed that non-communicating functions satisfy the design for test conditions of Sect. 6. Alphabets of  $P_i$  extended to satisfy the above requirements are denoted  $\Sigma_i^T$  and  $\Gamma_i^T$ .

For testing, it is a lot easier to deal with SXMSs rather with systems of them, hence a construction is proposed in [IH02] to convert a testable CSXMS  $S_n^{Ta}$  into an X-machine  $X_n^{Ta}$  computing the same function. A test set can be subsequently generated for  $X_n^{Ta}$  using the method described in Sect. 7 and applied with minor changes to the original testable CSXMS. The SXM  $X_n^{Ta}$  has its states defined as a cross-product of states in all SXMSs and the matrix  $EM$ ,  $(q_1, \dots, q_n, EM)$ ; a memory of  $X_n^{Ta}$  is

$$((m_1, in_1, out_1, s_1, g_1), \dots, (m_n, in_n, out_n, s_n, g_n)).$$

Relations of  $X_n^{Ta}$  are of the form  $(\phi_1 \dots \phi_n) \in CROSS(\Phi)$ .

**Theorem 9.1** [IH02] A SXM  $X_n^{Ta}$  constructed from a testable CSXMS  $S_n^{Ta}$  satisfies the design for test requirements of Sect. 6.

Definitions of testing compatibility (Definition 7.1) and a test set (Definition 7.2) can be applied to testing of CSXMS, where  $X_n^{Ta}$  plays the rôle of a specification.

Theorem 9.1 permits results of Sect. 7 to be applied to  $X_n^{Ta}$ . The resulting test set consists of a set of sequences of tuples from  $CROSS(\Sigma^{Ta})$ . Since attempting any of the communicating functions is accomplished using the same input  $a$ , a tester has to assume that the complete-testing assumption is satisfied and attempt a test sequence a number of times, until either the expected path is followed by an implementation or the a priori known number of attempts have been performed. It is worth noting that a CSXMS non-deterministically takes one of the possible sequences of communicating functions after every input without an explicit command from a tester. For this reason, [IH02] suggests not adding a special input  $a$  to communication functions and using a test set with all inputs  $a$  replaced by  $\epsilon$  and with tuples consisting entirely of  $\epsilon$  removed. A function performing these removals is denoted by *filter*. A modification of a testable specification  $S_n^{Ta}$  to use empty inputs instead of  $a$  will be denoted by  $S_n^T$  below and the equivalent of  $X_n^{Ta}$  will be denoted by  $X_n^T$ .

**Definition 9.3** Let  $S_n^{Ta}$  be a testable CSXMS,  $X_n^{Ta}$  a SXM constructed from it and  $S_n^T$ ,  $X_n^T$  be modifications of the two which use  $\epsilon$  instead of  $a$ . Let  $C$  be a set of SXMs which are nsxm testing compatible (Definition 7.1) with  $X_n^T$  and there is a known upper bound on the number of states in an associated automaton of  $Z' \in C$ . A set  $Y \subseteq (CROSS(\Sigma))^*$  is called a *csxm test set* if the following is satisfied:

$$\forall Z' \in C \text{ if for every } y \in Y \text{ there is a } g \in (CROSS(\Gamma^T))^* \text{ such that } y \| f_{S_n^T} \| g = y \| f_{Z'} \| g \text{ then } f_{Z'} = f_{S_n^T}.$$

### 9.1. Pre-requisites

A specification has to be testable (Definition 9.2) and restrictions imposed on a specification and an implementation in Sect. 7.1 have to be satisfied for  $X_n^T$  and an implementation. More precisely,

- A specification  $S_n^{Ta}$  is testable.
- An implementation can be modelled by a SXM with the same input ( $CROSS(\Sigma)$ ), output ( $CROSS(\Gamma^T)$ ) alphabets, memory sets and initial memory values as  $X_n^T$ . Additionally, an implementation has to have the same set of relations as  $X_n^T$ .
- An associate automaton of  $X_n^{Ta}$  is deterministic and minimal. Determinism can be ensured if all  $P_i$  of  $S_n^T$  are deterministic. Minimality cannot be directly related to the minimality of  $P_i$  and  $\Phi_i \cap \Phi_j = \emptyset$  ( $i \neq j$ ), because the state space of  $X_n^{Ta}$  contains not only the product of the state sets of CSXMs but also  $EM$ , so that some states of  $X_n^{Ta}$  may be unreachable.
- Both  $X_n^T$  and an implementation are completely defined. SXM  $X_n^T$  is completely defined if all  $P_i$  are completely defined.
- There is a known upper bound on the number of states in an implementation.
- There exists a testing context  $(V, U)$ , such that (1) a set of all relations of  $S_n^T$  is closed (Definition 6.1), output-distinguishable (Definition 6.3) and observable (Definition 6.4) w.r.t  $(V, U)$  and (2) a set of all non-communicating relations is input-complete (Definition 6.2) w.r.t  $(V, U)$ . Output-distinguishability and observability of communicating functions follows from the construction of  $S_n^T$ .

### 9.2. Generation of the test set

Test generation consists of the generation of a test set for  $X_n^{Ta}$  following the description in Sect. 7.2 and subsequent application of *filter* to it.

**Theorem 9.2** Under the conditions listed above,  $filter(t(Y_{m-n}))$  is a csxm test set (Definition 9.3).

### 9.3. Test set size and length

The maximal test size and length can be derived from results of Sect. 7.3, applied to test generation from  $X_n^{Ta}$ . The application of *filter* to these tests can reduce both the length and the number of tests, but the reduction depends on the particular system to test.

## 10. Conclusions

This paper described the X-machine testing method and its use for testing of different types of systems, both in terms of theory and practical outcomes.

On the theory side, the original X-machine testing method has been extended for testing non-deterministic software both for equivalence and conformance and also for testing of communicating systems (with specific restrictions on communications). The described extensions of the X-machine testing method done by different authors (i.e. [IpH00] vs. [HiH04]) are often complementary. For this reason, promising future work would be integration of these methods in order to realise all the benefits offered by these extensions. In particular, one might extend [HiH04] to (1) testing of non-deterministic implementations, (2) use no reset (following [HU02, IU99]) and (3) utilise the function cover idea (Sect. 5) to test functions together with testing of a transition diagram. Testing of communicating X-machines (Sect. 9) can also be extended to (1) handle communicating functions without using separate outputs from them and (2) perform a function cover of all non-communicating functions.

The X-machine testing method has also been extended to Harel statecharts [Bog00, BH02]. This has been made possible by substantially restricting statecharts, such as by prohibiting transitions in different states from carrying the same function and ensuring that statecharts do not execute a sequence of transitions without waiting for a command from a tester. Here future work involves the extension of the testing method to handle less constrained and non-deterministic statecharts.

On a practical side, the X-machine testing method has shown its applicability to large industrial problems (Sect. 4). Future work involves (1) using X-machines for testing of systems of a more diverse nature (such as for testing of communicating objects [BH05]) and (2) adapting it to testing in non-traditional software contexts. Some work on the latter has already been done [ThH03] in the context of Extreme Programming (XP).

## References

- [BGI03] Bălănescu T, Gheorghie M, Ipate F, Holcombe M (2003) Formal black box testing for partially specified deterministic finite state machines, *Found Comput Decis Syst* 28(1)
- [BH02] Bogdanov K, Holcombe M (2002) Testing from statecharts using the Wp method. In: *Proceedings of CONCUR'02 satellite workshop on formal approaches to testing (FATES)*, pp 19–33
- [Bee94] von der Beeck M (1994) A comparison of statecharts variants. *Lect Notes Comput Sci* 863:128–148
- [Ber91] Bernot G, Gaudel M, Marre B (1991) Software testing based on formal specifications: a theory and a tool. *Softw Eng J*, 6(6):387–405
- [BH05] Bogdanov K, Holcombe M (2005) Testing from object machines in practice. In: *Proceedings of UK software testing research III workshop (UK Test)*, pp 67–81
- [Bog00] Bogdanov K (2000) Automated testing of Harel's statecharts. PhD thesis, University of Sheffield, Sheffield
- [ChK93] Cheng K, Krishnakumar A (1993) Automatic functional test generation using the extended finite state machine model. In: *Proceedings of the 30th design automation conference*, Dallas, Texas, USA, ACM Press, New Orleans pp 86–91
- [Cho78] Chow T (1978) Testing software design modelled by finite state machines, *IEEE Trans Softw Eng* 4(3):178–187
- [Coh96] Cohen, D (1996) *Introduction to computer theory*, 2nd edn. Wiley, New York
- [DiF93] Dick J, Faivre A (1993) Automating the generation and sequencing of test cases from model-based specifications. In: Woodcock J, Larsen P (eds), *FME '93: industrial strength formal methods*, vol 670, *Lecture Notes Comput Sci* p 268–284. *Formal Methods Europe*. Springer, Berlin Heidelberg Newyork
- [Eil74] Eilenberg S (1994) *Automata, languages and machines*. vol. A. Academic Press, New York
- [FBK91] Fujiwara S, von Bochmann G, Khendek F, Amalou M, Ghedamsi A (1991) Test selection based on finite state models. *IEEE Trans Softw Eng* 17(6):591–603
- [Gur00] Gurevich Y (2000) Sequential abstract state machines capture sequential algorithms. *ACM Trans Comput Logic* 1(1):77–111
- [Hie98] Hierons R (1998) Adaptive testing of a deterministic implementation against a nondeterministic finite state machine. *Comput J* 41(5):349–355
- [Hie97a] Hierons R (1997) Testing from semi-independent communicating finite state machines with a slow environment. *IEE Proc Softw Eng* 144(5–6):291–295
- [Hie97b] Hierons R. M. (1997) Testing from a finite state machine: extending invertibility to sequences. *COMPJ: The Comput J* 40(4) pp 220–230
- [HiH00] Hierons R, Harman M (2000) Testing conformance to a quasi-non-deterministic stream X-machine. *Formal Asp Comput* 12(6):423–442
- [HiH04] Hierons R, Harman M (2004) Testing conformance of a deterministic implementation against a non-deterministic stream X-machine. *Theor Comput Sci* (in press)
- [HIG95] Holcombe M, Ipate F, Grondoudis A (1995) Complete functional testing of safety-critical systems. *Proceedings of the 2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies*, Daytona Beach, Florida, USA, Elsevier, Oxford, pp 199–204
- [Hol88] Holcombe M (1988) X-machines as a basis for dynamic system specification. *Softw Eng J* 3:69–76
- [HaN96] Harel D, Naamad A (1996) The STATEMATE semantics of statecharts. *ACM Trans Softw Eng Methodol* 5(4):293–333

- [HoI98] Holcombe M, Ipatе F (1998) Correct systems: Building a business process solution. Springer, Berlin Heidelberg Newyork
- [HU02] Hierons R, Ural H (2002) Reduced length checking sequences. IEEEETC: IEEE Trans Comput 51:1111–1117
- [IpH96] Ipatе F, Holcombe M (1996) Another look at computability. Informatica 20:359–372
- [IpH97] Ipatе F, Holcombe M (1997) An integration testing method that is proved to find all faults. Int J Comput Math 63:159–178
- [IpH98a] Ipatе F, Holcombe M (1998) A method for refining and testing generalized machine specifications. Int J Comput Math 68:197–219
- [IpH98b] Ipatе F, Holcombe M (1998) Specification and testing using generalized machines: a presentation and a case study. Softw Test Verification Rel 8:61–81
- [IpH00] Ipatе F, Holcombe M (2000) Generating test sequences from non-deterministic generalized stream X-machines. Formal Asp Comput 12(6):443–458
- [IH02] Ipatе F, Holcombe M (2002) Testing conditions for communicating stream X-machine systems. Formal Asp Comput 13(6):431–446
- [IpH02a] Ipatе F, Holcombe M (2002) An integrated refinement and testing method for stream X-Machines. Appl Algebra Eng Commun Comput 13(2):67–91
- [IpH02b] Ipatе F, Holcombe M (2002) Testing conditions for communicating stream X-machine Systems. Formal Asp Comput 13(6):431–446
- [Ipa03] Ipatе F (2003) On the minimality of stream X-machines. Computer J 46(3) pp 295–306
- [Ipa04] Ipatе F (2004) Complete deterministic stream X-machine testing. Formal Asp Comput 16(4):374–386
- [IU99] Inan K, Ural H (1999) Efficient checking sequences for testing finite state machines. Inf Softw Technol 41:799–812
- [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: Mastorakis N (ed) Systems and control: theory and applications, World Scientific and Engineering Society Press, Athens, pp 163–171
- [LuBP94] Luo G, von Bochmann G, Petrenko A (1994) Test selection based on communicating nondeterministic finite-state machines using a generalized WP-method. IEEE Trans Softw Eng 20(2):149–162
- [LuPB94] Luo G, Petrenko A, von Bochmann G (1994) Selecting test sequences for partially specified nondeterministic finite state machines. In: Proceedings of IFIP 7th international workshop on protocol test systems, Japan, pp 95–110
- [LeY96] Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines – A survey. Proc IEEE 84(8):1090–1123
- [Omg03] OMG (2003) OMG Unified Modeling Language specification, version 1.5 <http://www.omg.org/technology/documents/formal/uml.htm>
- [OsB89] Ostrand T, Balcer M (1989) The category-partition method for specifying and generating functional tests. Commun ACM 31(6):667–686
- [PY02] Petrenko A, Yevtushenko N (2002) Queued testing of transition systems with inputs and outputs. In: Proceedings of CONCUR'02 Satellite workshop on formal approaches to testing (FATES), Brno, Czech Republic, pp 79–93
- [PeYB96] Petrenko A, Yevtushenko N, von Bochmann G (1996) Testing deterministic implementations from nondeterministic FSM specifications. In: Proceedings of 9th international workshop on testing of communicating systems (IWTCS'96), pp 125–140
- [PYvBD96] Petrenko A, Yevtushenko N, von Bochmann G, Dssouli R (1996) Testing in context: framework and test derivation. Comput Commun 19:1236–1249
- [RDT95] Ramalingam T, Das A, Thulasiraman K (1995) On testing and diagnosis of communication protocols based on the FSM model. Comput Commun 18(5):329–337
- [ThH03] Thomson C, Holcombe M (2003) Applying XP ideas formally. In: Proceedings of 1st south-east european workshop on formal methods (SEEFM'03), pp 57–71
- [Tre96] Tretmans J (1996) Test generation with inputs, outputs and repetitive quiescence. Softw Concepts Tools 17(3):103–120
- [Van02] Vanak S (2002) Complete functional testing of hardware descriptions. PhD thesis, University of Sheffield, Sheffield
- [Vas73] Vasilevskii M (1973) Failure diagnosis of automata. Cybernetics, Plenum publ Corporation, Newyork 4:653–665
- [Wal93] Wang C, Liu M (1993) Generating test cases for EFSM with given fault models. In: Proceedings of IEEE INFOCOM '93, vol 2. San Francisco, IEEE pp 774–781

*Received June 2004*

*Revised March 2005*

*Accepted March 2005 by J. Derrick, M. Harman and R. M. Herons*

*Published online 24 January 2006*