Testing conformance of a deterministic implementation against a non-deterministic stream X-machine

R. M. Hierons [a] M. Harman [a]

[a] *Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK.*

keywords: Stream X-machines, testing, non-determinism, conformance, deterministic implementation.

Abstract

Stream X-machines are a formalisation of extended finite state machines that have been used to specify systems. One of the great benefits of using stream X-machines, for the purpose of specification, is the associated test generation technique which produces a test that is guaranteed to determine correctness under certain design for test conditions. This test generation algorithm has recently been extended to the case where the specification is non-deterministic. However, the algorithms for testing from a non-deterministic stream X-machine currently have limitations: either they test for equivalence, rather than conformance or they restrict the source of non-determinism allowed in the specification. This paper introduces a new test generation algorithm that overcomes both of these limitations, for situations where the implementation is known to be deterministic.

1 Introduction

Many systems can be modelled by finite state machines. However, if the system's specification requires memory, then an extended form of finite state machine is required. The stream X machine is just such a form of extended finite state machine. A software development approach is associated with stream X-machines. Here a set of trusted components are integrated to form a larger system, where communication between the components is modelled using a shared memory. The approach allows test data to be constructed from a model of such a component–based system. The test data is then applied to the implementation.

The most striking aspect of the stream X machine approach is the, at first, implausible-sounding claim that the set of test data constructed from the model is sufficient to *guarantee* the correctness of the implementation. This sounds implausible because it seems to contradict Dijkstra's oft quoted aphorism, which appears, *inter alia*, in his book (co-authored with Dahl and Hoare [8]):

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

How could passing a finite set of tests ever provide a *guarantee* of a system's correctness?

The salient point here is that the approach does *not* guarantee that the implementation is correct if it should turn out that the trust placed in the 'trusted components' is a misplaced trust. That is, the approach guarantees that the *integration* of the trusted components is correct, on the *assumption* that the trusted components are, themselves, correct. In this way the stream X machine approach can be regarded as an instance of Gaudel's [12] testing framework, in which formal proof discharges one part of the correctness demonstration, while testing is used to discharge the remaining part. The combination of formal proof and testing thereby establishes the overall correctness of the system under consideration.

For stream X machines, a full correctness guarantee of the entire system would require the proof of correctness for the trusted components in addition to the results which show that the implementation passes all the tests constructed from the stream X machine model.

Since reliance on 'trusted components' is an increasing feature of software development, both by necessity and design, the stream X machine approach offers a soundly-based, yet practicable technique by which the correctness issue can be split into integration–correctness concerns and component–correctness concerns. As such, the approach allows one to guarantee correctness of the implementation concerns, in isolation, simply by passing a set of test data. This is a significant advantage of the approach, and has been the primary motivation for its study (for example see [3–6,19,21]).

The components used to construct the implementation could have been developed from smaller (trusted) components using the Stream X-machine approach. Thus a system could be built from basic components through a sequence of refinement and testing phases.

The model of testing from a stream X machine is one in which tests are generated from a state based model (the stream X machine), and are applied to an implementation which, it is hoped, respects the model. Traditionally, only

deterministic stream X-machines have been used for the purpose of describing specifications. This was largely because the stream X-machine test technique was only applicable to deterministic stream X-machines.

However, the restriction to deterministic stream X machines is clearly a significant barrier to its wider uptake. It is part of the nature of a specification to want to leave certain paths open to the designer and implementor of the system. A favourite mechanism by which this is achieved is that of making the specification non-deterministic. That is, the specifier of a system, merely indicates that one of several possibilities should be implemented. This leaves the implementor free to choose that which is the most efficient or practical according to a set of concerns and criteria, the detail of which is unknown or unimportant at the specification level. A technique that is capable of generating test data from non-deterministic stream X machines is therefore an important research goal.

Recently, the test technique has been extended to allow non-determinism [16,24]. However, each piece of published work on testing from a non-deterministic stream X-machine suffers from at least one of the following restrictions:

(1) The test generation algorithms assume that the notion of correctness used is equivalence [24]. Thus, the test determines whether the set of traces in the *implementation under test (IUT)* is identical to that in the specification. However, when the specification is non-deterministic, often the appropriate notion of correctness is conformance: the set of traces in the IUT is contained within the set of traces of the specification. Where conformance is the appropriate form of correctness, algorithms that test for equivalence are not applicable.
(2) The algorithms limit the source of non-determinism in the specification [16], thus restricting the set of specifications to which the approach may be applied.

This paper extends the current work by considering the problem of testing a deterministic implementation for conformance to a general non-deterministic stream X-machine. The case in which the specification is non-deterministic and the implementation is deterministic is highly relevant; while most implementations are deterministic, non-determinism aids abstraction and thus is appropriate for specifications. Here the appropriate notion of correctness is conformance rather than equivalence and thus this case is not covered by current approaches to non-determinism. A further extension to the work by Hierons and Harman [16] is provided by weakening the design for test conditions in a similar manner to the changes made by Ipate and Holcombe [24].

When testing from a stream X-machine M it is normal to assume that the IUT I behaves like some *unknown* stream X-machine M_I. Interestingly, in

the case considered in this paper, I may conform to M even if M_I and M have significantly different structures. This contrasts with problems previously considered. An important consequence of this observation is that the traditionally-used W-method cannot be applied. In its place a test procedure, based on the notion of state counting, is introduced. State counting has previously been used for testing from a Non-deterministic Finite State Machine (see, for example, [31,37]).

The rest of this paper is structured as follows. Section 2 briefly reviews the testing of state based systems. Section 3 provides preliminary material and gives an example. Section 4 defines the design for test conditions used in this paper. Section 5 characterises conformance in terms of a relationship between languages defined by M_I and M. Section 6 introduces the test process that allows the tester to decide whether a word is a member of the language defined by the stream X-machine M_I, that represents the implementation, through black-box testing. Section 7 considers the problem of finding sequences to reach and distinguish states of M; this problem is significantly altered by the conditions considered here. Based on this, Section 8 introduces an algorithm that produces a test that is guaranteed to determine correctness under the design for test conditions. Section 9 then discusses possible future work and finally, Section 10 draws conclusions.

2 Background and Motivation

2.1 State based systems

Many systems have a persistent internal state and such systems are often specified using state-based languages such as Statecharts [13] and SDL [26]. These languages specify a system in terms of a finite set of logical states, an internal store, and transitions between the states, each transition being labelled with an operation that may change the store. Typically, the logical state is used to indicate which sequences of operations are currently possible while the store is used to hold additional information. Thus, the logical states and transitions between them specify the control structure, while the operations that label the transitions specify the data processing.

Consider, for example, a video recorder (VCR). A model of a VCR might have logical states such as one representing the VCR being in play mode and another representing the VCR being paused. There might also be other data, such as a counter to state how long the currently-loaded cassette has been playing and information about the configuration settings of the VCR. This additional data forms part of the internal store. Such a state-based view

of the behaviour of a VCR is highly amenable to state-based modeling and reasoning.

State-based specification languages have been used for a variety of systems. SDL is used for the specification of communications protocols while Statecharts are widely used for the specification of reactive systems and now form part of the Unified Modeling Language (UML). Specifications written in such languages can usually be thought of as extended finite state machines (EFSMs).

Model-based languages such as Z and VDM have also been used for specifying systems that have an internal state. Interestingly, it has been recognised that it is useful to devise a logical state structure, and thus produce an EFSM, when testing from such a specification (see, for example, [9,10,14,33]). The presence of a logical state structure provides a number of benefits when testing. For example, it helps in the process of finding a sequence of inputs or events that set up the state in order for a test to be applied.

The wide reliance upon state-based models for specification, design and reasoning about systems has led to a significant research effort concerned with the verification of state-based systems. One of the primary concerns for this state-based verification research agenda has been the question of how best to test state-based systems.

2.2 Testing state based systems

Testing is a process in which the implementation under test (IUT) is provided with sequences of input values and the resultant behaviours are observed and checked against the specification. Testing is often divided into at least the following three stages:

(1) Unit testing: the individual components of the IUT are tested against their specifications.
(2) Integration testing: the interaction between these components is checked.
(3) System testing: the overall functionality of the system is tested against the requirements. This phase will often involve users.

When testing from an EFSM, it is sometimes possible to apply techniques that have been developed for testing from a finite state machine (or transducer). There is a wide range of such techniques (see, for example, [1,15,17,28,34,35]). However, in order to apply such techniques, it is necessary to produce a finite state machine (FSM) from the given EFSM specification. This FSM could be produced using one of the following approaches.

(1) Expand out the internal store.
(2) Abstract away the internal store.

Where the internal store is infinite, it is not possible to produce an FSM by expanding out the store. Even where the store is finite, this process leads to a combinatorial explosion. Thus, for many EFSM specifications, it is not practical to expand out the internal store. If the store is abstracted away the resultant sequences need not be feasible in the original EFSM since the abstraction process removes the preconditions from the transitions (see, for example, [18,36]). Further, it is often difficult to relate the fault coverage of the resultant FSM to that of the EFSM. Thus, each of these approaches has limitations.

When testing from an FSM M, it is often possible to produce a checking experiment: a test that is guaranteed to determine correctness under certain conditions (see, for example, [7,15,17,30,34]). Typically, it is assumed that the IUT behaves like some unknown FSM M' that has the same input and output alphabets as M and no more than m states for some predefined m. Thus, where it is practical to produce an FSM model from the specification, there exist test generation techniques that provide strong guarantees regarding the fault-detecting ability of the resultant test sequence.

2.3 Stream X-machines

X-machines were introduced by Eilenberg [11]. Later, Holcombe [19] proposed their use as a specification formalism. The stream X-machine formalism specifies a system as an EFSM. Stream X-machines provide a convenient standard formalism within which issues such as test generation may be considered. Further, they have been used to specify a range of systems (see, for example, [4,19–21,27]). Results regarding testing from a stream X-machine can be applied when testing from specifications written in other state based languages such as Statecharts (see, for example, [6]).

Associated with stream X-machines is a development and testing philosophy [21]. Under this philosophy, it is assumed that the system is built from a set of trusted components. These components may have been tested in a previous phase, such as unit testing, or they might be imported from a library. System development could proceed through a sequence of steps, each of which involves building larger components from smaller components that have already been developed (see, for example, [21]). Thus, the testing problem reduces to checking that these components have been combined in the correct way and so it might be seen as an approach to integration testing. This philosophy leads to methods that generate tests that are guaranteed to determine correctness

under certain design for test conditions (see, for example, [4,16,19–24,27]).

When testing from a stream X-machine M, it is normal to assume that the IUT behaves like some *unknown* stream X-machine M_I. This makes it possible to formally reason about test effectiveness. In testing it is desirable to apply a set of input sequences that, between them, determine whether the unknown model M_I is a correct implementation of M. This paper introduces a new algorithm that produces a test that, under certain design for test conditions, determines whether a deterministic implementation conforms to a specification in the form of a non-deterministic stream X-machine.

It is worth noting that, even where it is practical to produce an FSM from an EFSM by expanding out the store, the test resulting from applying FSM based techniques will normally be much larger than that produced using the stream X-machine methods. This is because the stream X-machine test techniques utilise the belief that the individual components are correct. They avoid the state explosion associated with expanding out the store and may be applied when the store is infinite.

3 Preliminaries and Example

3.1 Finite automata

A *finite automaton (FA)* N is defined by a tuple $(S, s_0, Z, \delta, \Gamma)$ in which S is a finite set of states, $s_0 \in S$ is the initial state, Z is the finite input alphabet, δ is the state transfer relation of type $S \times Z \leftrightarrow S$, and $\Gamma \subseteq S$ is the set of final states. If N receives an input $z \in Z$ when in state $s \in S$ it moves to some state in the set $\delta(s, z)$. Note that given sets A and B, $A \leftrightarrow B$ denotes the set of relations between A and B and so may be considered to be equivalent to $A \times B$. Further, if relation r has type $A \leftrightarrow B$ and $a \in A$ then $r(a)$ denotes the set of elements of B related to a under r: $r(a) = \{b \in B \mid r(a, b)\}$. The relation δ may be extended to take an input sequence, giving the relation δ^* defined below.

Definition 1 *Let ϵ denote the empty sequence and $z \in Z$, $\overline{z} \in Z^*$. The following define δ^*.*

$$\delta^*(s, \epsilon) = \{s\}$$

$$\delta^*(s, \overline{z}z) = \{s' \mid \exists\, s''.s'' \in \delta^*(s, \overline{z}) \wedge s' \in \delta(s'', z)\}$$

Throughout this paper, a variable name with a line over it will denote a sequence. The FA N defines a language $L(N)$, of words that can take N from its initial state to some final state, in the following way.

Definition 2 *Given a FA* $N = (S, s_0, Z, \delta, \Gamma)$ *the language* $L(N)$ *is defined as* $\{\overline{z} \in Z^* \mid \delta^*(s_0, \overline{z}) \cap \Gamma \neq \varnothing\}$.

Further, given a state s of N, there is a corresponding language formed from words that take N from s to a final state.

Definition 3 *Given a FA* $N = (S, s_0, Z, \delta, \Gamma)$ *and state* $s \in S$ *the language* $L_N(s)$ *is defined as* $\{\overline{z} \in Z^* \mid \delta^*(s, \overline{z}) \cap \Gamma \neq \varnothing\}$.

Clearly $L(N) = L_N(s_0)$.

A FA is *deterministic* if for all $s \in S$ and $z \in Z$ there is at most one possible next state: $\forall\, s \in S, z \in Z. \mid \delta(s, z) \mid \leq 1$. Two FA are *equivalent* if they define the same language. Given FA N, there is some equivalent deterministic FA [32]. A deterministic FA (DFA) is *minimal* if there is no equivalent DFA with fewer states. Any FA may be rewritten to an equivalent minimal DFA [29]. It will thus be assumed that any FA considered is deterministic and minimal.

3.2 Stream X-machines

A stream X-machine is a form of extended finite state machine in which there is a set of states, the transitions between states are labelled with relations, and there is an internal memory. More formally, a stream X-machine is defined by a tuple $(In, Out, S, Mem, \Phi, F, s_0, m_0, \Gamma)$ [21] in which:

- In is the input alphabet.
- Out is the output alphabet.
- S is the finite set of states.
- Mem is the memory. Mem need not be finite.
- Φ is a set of processing relations, each having type $Mem \times In \leftrightarrow Out \times Mem$.
- F is the next state relation of type $S \times \Phi \leftrightarrow S$.
- $s_0 \in S$ is the initial state.
- $m_0 \in Mem$ is the initial memory value.
- Γ is the set of final states.

Essentially, the state transition structure of stream X-machine M determines a set L of sequences of relations from Φ^*: the sequences that label walks from the initial state of M to some final state of M. Each of these sequences defines a relationship between input sequences and output sequences. The behaviour defined by M is the union of the relationships (of type $In^* \leftrightarrow Out^*$) defined

by the sequences in L. This specified behaviour will be formally defined in Section 3.4 and will be illustrated by an example in Section 3.3. Note that traditional definitions of stream X-machines limit the sets In and Out to being finite. However, it transpires that the results regarding test generation do not require these restrictions to be in place. Therefore, in this paper, In and Out are allowed to be infinite.

The set Φ is often called the *type* of M. This set denotes the set of relations from which M is built. Typically, each element of Φ specifies components that may be used in the construction of the implementation. Since the philosophy behind stream X-machine test techniques is that the IUT is built from components that are known (or trusted) to be correct, the set Φ places restrictions on the IUT.

Observe that the memory Mem need not correspond to the notion of the memory of a program. Rather, Mem models the passing of values between components as a (possibly infinite) memory. Thus, Mem might be formed from tuples, where each element of the tuple corresponds to either a global variable or a parameter that may be passed between two relations in Φ.

The next state relation F can be extended, to take sequences from Φ^*, to form the relation F^*. It is possible to allow a set of initial states, rather than a single initial state. However, allowing a set S_I of initial states does not significantly affect the test generation problem: a test may be devised by combining those produced for each possible initial state from S_I. Thus, to simplify the explanation, the definition of a stream X-machine will include one initial state only.

3.3 Example

This section introduces an example specification of a stream X machine for a simple calculation system. The example will be used throughout the paper to illustrate the approach to testing from non-deterministic stream X machines. The example has been chosen to illustrate the central issues with non-deterministic stream X machines. In order to do this, the example must contain at least two states which are not deterministically reachable and two states which are not pairwise distinguishable. These terms will be formally defined later. Informally, what this entails, is a stream X machine where there are two states for which there is no sequence of inputs which is guaranteed to reach them (not deterministically reachable) and there are two states for which there is no input sequence which is guaranteed to trigger an output that distinguishes them. Each of these two properties make testing harder and complicates the example.

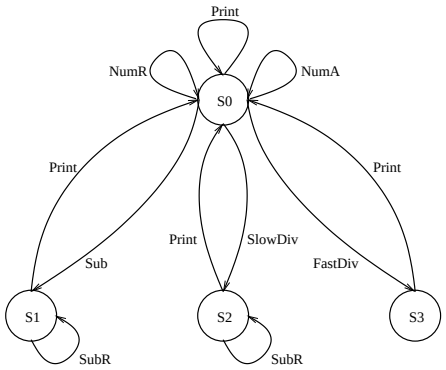

Fig. 1. A Simple Non–deterministic Division Calculator

The challenge is therefore to find a suitable example which has these properties, but which is not so complex as to lose its expository value. The example presented here is simplified in order to ensure that it adequately illustrates the definitions, concepts and testing process. The example is contrived in the sense that it is unlikely that such a simplistic calculating device would be built in practice, but is not so contrived that the design choices involved are without intuitive foundation.

The state structure of the calculator stream X-machine is pictured on Figure 1. In what follows, its input, output, memory and operations will be formally defined using the Z notation. The calculator has BUTTONS, Lights and a numeric keypad. Buttons are input devices used to select a particular behaviour. The calculator has a memory which consists of three non-negative integers: the accumulator (A), register (R) and index (I). Initially these three integers are set to zero. The relations used in the machine will now be described.

There are six buttons, in the set BUTTONS. The $\mathcal{S}$ button, is used to request a subtraction operation, which subtracts the current value of the register from the current value of the accumulator, storing the result in the accumulator. The buttons represent inputs to the system. The $\mathcal{R}$ button, is used to request a single repetition of the subtraction operation. The $\mathcal{D}$ button, is used to request the division operation. The $\mathcal{P}r$ button, is used to request the print operation. The $\mathcal{N}A$ button, is used to indicate that a numeric input is to be stored in the Accumulator. The $\mathcal{N}R$ button, is used to indicate that a numeric input is to be stored in the register. It is not possible to directly store a value in the index. The $\mathcal{N}A$ and $\mathcal{N}R$ button are used in conjunction with a numeric keypad which allows the user to enter a single non-negative number.

The calculator has four lights LSub, LSubR, LSlowDiv and LFastDiv, corresponding to the operations, Sub, SubR, SlowDiv and FastDiv. Each of these lights is illuminated when the corresponding operation is invoked. There is also an underflow light LUnderflow, which is illuminated when an attempt

is made to evaluate an expression which would lead to a negative result.

In addition to the lights, there is also a simple screen output device, which is capable of displaying up to two non-negative integers in the range storeable by the accumulator and register. This will be assumed to be the natural numbers, $\mathbb{N}$.

The operations NumR and NumA cause input to be read from the numeric keypad. The NumR operation is triggered by $\mathcal{N}R$, while the NumA operation is triggered by the $\mathcal{N}A$ button. When the NumR operation is executed the number previously read into the numeric keypad is stored in the register (R) and the value of the number read in is displayed on the screen. When the NumA operation is executed, the number previously read into the numeric keypad is stored in the accumulator (A) and the value of the number read in is displayed on the screen.

Finally, at any point in the execution of the machine, the user can press the $\mathcal{P}r$ button, causing the Print operation to be executed. This causes the value currently stored in the accumulator and register to be displayed on the screen.

The operations, Sub, SubR, SlowDiv and FastDiv perform simple computations on the values stored in the memory. The Sub operation, responds to the $\mathcal{S}$ button, storing the results of subtracting the register from the accumulator (or zero if this would lead to a negative result).

The SubR operation, responds to the $\mathcal{R}$ button. It can be used in conjunction with the SlowDiv operation to achieve division by repeated subtraction. If the value of the accumulator is greater than or equal to the value of the register, then the SubR operation increases the index register by one and subtracts the register contents from the accumulator contents. This will happen if additional iterations are required to compute the integer division result by repeated subtraction. If the accumulator value is less then the register value then the operation does not affect any of the memory values but illuminates the LUnderflow light. This illumination signifies the end of a sequence of applications of the SubR operation. In this way, should the user trigger repeated applications of the SubR operation, by repeatedly pressing the $\mathcal{S}$ button, the machine will compute integer division by repeated subtraction.

The FastDiv operation, responds to the $\mathcal{D}$ button. It stores the result of dividing the contents of the accumulator by the contents of the register using integer division and illuminates the LFastDiv light.

The SlowDiv operation, also responds to the $\mathcal{D}$ button. It affects neither the accumulator nor the register, but stores zero in the index and illuminates the LSlowDiv light. The SlowDiv operation establishes a logical state in which it is possible to compute the result of dividing the contents of the accumulator

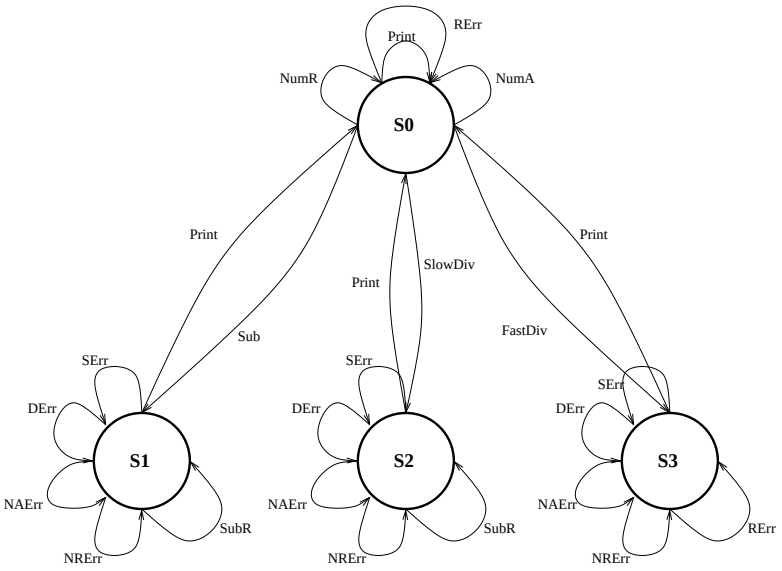

Fig. 2. The Completely Specified Non–deterministic Division Calculator

by the contents of the register.

This is achieved (albeit slowly) by repeated subtraction; the user must repeatedly invoke the SubR operation until the LUnderflow light is illuminated.

In each state there is a set of operations $\{\text{SErr}, \text{DErr}, \text{PErr}, \text{RErr}, \text{NAErr}, \text{NRErr}\}$ which have the 'no operation' effect (other than to light the error light, LError). Adding these operations to the state diagram, complicates the diagram (see Figure 2), but does not affect the essential core structure of the stream X machine specification depicted in Figure 1.

The specification is non-deterministic because either the FastDiv operation or the SlowDiv operation can be triggered by the $\mathcal{D}$ button. A deterministic implementation must choose between these two.

The above definitions of operations are made more formal and are related back to the definition of a stream X machine in the Z specification which follows.

$$BUTTONS ::= \mathcal{R} \mid \mathcal{D} \mid \mathcal{S} \mid \mathcal{P}r \mid \mathcal{N}A \mid \mathcal{N}R$$

$$\mathsf{Lights} ::= \mathsf{LSub} \mid \mathsf{LSubR} \mid \mathsf{LSlowDiv} \mid \mathsf{LFastDiv} \mid \mathsf{LUnderflow} \mid \mathsf{LError}$$

The memory (the *Mem* component of the stream X Machine as defined in Section 3.2), consists of three components, the accumulator, A, the register, R and the index I.

Memory

$A : \mathbb{N}$
$R : \mathbb{N}$
$I : \mathbb{N}$

The initial state of the memory (m_0 in the definition in Section 3.2) is defined by the Schema below:

InitialMemory

$\Delta Memory$

$A' = 0$
$R' = 0$
$I' = 0$

In the following schemas, the input events (decorated with a ?) correspond to the pressing of buttons, while the output events (decorated with !) correspond to the illumination of lights or the display of accumulator and register values on the screen. In terms of the definition of a stream X machine presented in Section 3.2, the input events form the set In, while the output events form the set Out.

The user functions PRINT, NUMR, NUMA, SUB, SUBR, SLOWDIV and FASTDIV, form the set of operations (the set Φ in the definition in Section 3.2) and are defined as follows:

PRINT

$\Xi Memory$
$b? : BUTTONS$
$r! : \mathbb{N} \times \mathbb{N}$

$b? = \mathcal{P} r$
$r! = (A, R)$

Sub

$b? : BUTTONS$
$\Delta Memory$
$r! : \mathsf{Lights}$

$b? = \mathcal{S}$
$(A \geq R \land A' = A - R \ \lor \ A < R \land A' = 0)$
$R' = R$
$I' = I$
$r! = \mathsf{LSub}$

SubR

$b? : BUTTONS$
$\Delta Memory$
$r! : \mathsf{Lights}$

$b? = \mathcal{S}$
$(A - R < 0 \ \land A' = I \ \land R' = R \ \land I' = I \ \land r! = \mathsf{LUnderflow}$
$\lor$
$A - R \geq 0 \ \land A' = A - R \ \land R' = R \ \land I' = I + 1 \ \land r! = \mathsf{LSubR})$

NumR

$u? : BUTTONS \times \mathbb{N}$
$\Delta Memory$
$r! : \mathbb{N}$

$\exists\, i : \mathbb{N} \bullet u? = (\mathcal{N} R, i)$
$A' = A$
$R' = i$
$I' = I$
$r! = i$

NumA

$u? : BUTTONS \times \mathbb{N}$
$\Delta Memory$
$r! : \mathbb{N}$

$\exists\, i : \mathbb{N} \bullet u? = (\mathcal{N} A, i)$
$A' = i$
$R' = R$
$I' = I$
$r! = i$

FastDiv

$b? : BUTTONS$
$\Delta Memory$
$r! : \mathsf{Lights}$

$b? = \mathcal{D}$
$(R > 0 \land A' = A/R) \lor (R = 0 \land A' = 0)$
$R' = R$
$I' = I$
$r! = \mathsf{LFastDiv}$

SlowDiv

$b? : BUTTONS$
$\Delta Memory$
$r! : \mathsf{Lights}$

$b? = \mathcal{D}$
$A' = A$
$R' = R$
$I' = 0$
$r! = \mathsf{LSlowDiv}$

In addition to the user functions above, there is a set of six 'error' functions which are triggered when the user attempts to invoke a function which has no effect. The presence of these functions makes the specification completely specified.

SErr

$b? : BUTTONS$
$\Xi Memory$
$r! : \mathsf{Lights}$

$b? = \mathcal{S}$
$r! = \mathsf{LError}$

DErr

$b? : BUTTONS$
$\Xi Memory$
$r! : \mathsf{Lights}$

$b? = \mathcal{D}$
$r! = \mathsf{LError}$

PErr

$b? : BUTTONS$
$\Xi Memory$
$r! : \mathsf{Lights}$

$b? = \mathcal{P}r$
$r! = \mathsf{LError}$

RErr

$b? : BUTTONS$
$\Xi Memory$
$r! : \mathsf{Lights}$

$b? = \mathcal{R}$
$r! = \mathsf{LError}$

NAErr

$u? : BUTTONS \times \mathbb{N}$
$\Xi Memory$
$r! : \mathsf{Lights}$

$\exists\, i : \mathbb{N} \bullet u? = (\mathcal{N}A, i)$
$r! = \mathsf{LError}$

NRErr

$u? : BUTTONS \times \mathbb{N}$
$\Xi Memory$
$r! : \mathsf{Lights}$

$\exists\, i : \mathbb{N} \bullet u? = (\mathcal{N}R, i)$
$r! = \mathsf{LError}$

3.4 Properties of stream X-machines

This section will describe a number of properties of stream X-machines that will be used throughout the paper. It will also define the semantics of stream X-machines.

A stream X-machine M can be represented by a finite automaton, called the associated automaton, that is defined below. Essentially, the associated automaton inherits the state and transition structure of the stream X-machine but has no internal memory.

Definition 4 *Given stream X-machine* $M = (In, Out, S, Mem, \Phi, F, s_0, m_0, \Gamma)$, *the* associated automaton $A(M)$ *is* $(S, s_0, \Phi, F, \Gamma)$.

The stream X-machine M is *minimal* if $A(M)$ is minimal. When looking at the problem of testing from a stream X-machine it is normal to assume that every state is a final state and thus $\Gamma = S$ [21]. This is not usually a restriction when considering interactive systems. Since any non-deterministic finite automaton can be rewritten to form an equivalent minimal deterministic FA (DFA), it will be assumed that $A(M)$ is a minimal DFA and thus that F is a function.

Given a sequence $\overline{f}$ of elements from Φ, $\|\overline{f}\|$ will denote the relation of type $Mem \times In^* \leftrightarrow Out^* \times Mem$ induced by $\overline{f}$. Essentially, $\|\overline{f}\|$ corresponds to the possible results of executing the sequence of relations from $\overline{f}$ in the given order.

Definition 5 *Given a sequence* $\overline{g} \in \Phi^*$, $\overline{g}$ *induces the relation* $\|\overline{g}\|$, *of type* $Mem \times In^* \leftrightarrow Out^* \times Mem$, *defined by the following in which* $f \in \Phi$ *and* $\overline{f} \in \Phi^*$.

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

$$\begin{aligned}\|\overline{f}f\| = \{&((m, \overline{x}x), (\overline{y}y, m')) \mid \exists\, m'' \in Mem.((m, \overline{x}), (\overline{y}, m'')) \in \|\overline{f}\| \\ &\wedge((m'', x), (y, m')) \in f\}\end{aligned}$$

Consider, for example, the sequence $<$ NumR, NumA $>$ of operations from the calculator example. The first operation has an input consisting of an integer x_1 and the pressing of the $\mathcal{N}R$ button. It updates the register and outputs the value x_1. The second operation has an input consisting of an integer x_2 and the pressing of the $\mathcal{N}A$ button. It updates the accumulator and outputs the value x_2. Let the memory with $A = a$, $R = r$ and $I = i$ be denoted by the tuple (a, r, i). Thus, the following is the relation defined by $<$ NumR, NumA $>$.

$$\begin{aligned}&\| < \text{NumR}, \text{NumA} > \| = \\ &\{(((a, r, i), < (\mathcal{N}R, x_1), (\mathcal{N}A, x_2) >), (< x_1, x_2 >, (x_2, x_1, i))) \mid \\ &a \in \mathbb{N} \wedge r \in \mathbb{N} \wedge i \in \mathbb{N} \wedge x_1 \in \mathbb{N} \wedge x_2 \in \mathbb{N}\}\end{aligned}$$

Since a stream X-machine starts with an initial memory m_0, $\overline{f}$ defines a relation $\langle \overline{f} \rangle$ between input sequences and output sequences. This is formed by restricting the relation $\|\overline{f}\|$ to the case where the initial memory is m_0 and then abstracting away the final memory.

Definition 6

$$\langle \overline{f} \rangle = \{(\overline{x}, \overline{y}) \mid \exists\, m \in Mem.((m_0, \overline{x}), (\overline{y}, m)) \in \|\overline{f}\|\}$$

The calculator starts with memory $(0, 0, 0)$. Thus $\langle < \textsc{NumR}, \textsc{NumA} > \rangle$ is as follows:

$$\langle < \textsc{NumR}, \textsc{NumA} > \rangle = \\ \{(< (\mathcal{N}R, x_1), (\mathcal{N}A, x_2) >, < x_1, x_2 >) \mid x_1 \in \mathbb{N} \wedge x_2 \in \mathbb{N}\}$$

The stream X-machine M can be seen as defining a relation between input sequences and output sequences. An input sequence $\overline{x}$ is related to an output sequence $\overline{y}$ if some sequence of consecutive arcs, from the initial state of M to a final state of M, gives a sequence of relations that allows $\overline{y}$ to be produced in response to $\overline{x}$ when the initial memory is m_0. The set of sequences of arcs from the initial state of M to a final state of M defines the regular language $L(A(M))$ and each sequence $\overline{f} \in L(A(M))$ induces a relation $\langle \overline{f} \rangle$ of type $In^* \leftrightarrow Out^*$. More formally, M defines a relation, denoted $\lfloor M \rfloor$, of type $In^* \leftrightarrow Out^*$ defined in the following way.

Definition 7

$$\lfloor M \rfloor = \bigcup_{\overline{f} \in L(A(M))} \langle \overline{f} \rangle$$

Definition 8 *Given a relation R of type $A \leftrightarrow B$, $dom\, R$ denotes the set of values in A related to values in B under R.*

$$dom\, R = \{a \in A \mid \exists\, b.b \in B \wedge (a, b) \in R\}$$

The stream X-machine M has an input domain: the set of input sequences that are related to output sequences under $\lfloor M \rfloor$.

Definition 9 *Given a stream X-machine M, the* input domain *of M, denoted $dom\, M$, is defined by:*

$$dom\, M = \{\overline{x} \in In^* \mid \exists\, \overline{y}.\overline{y} \in Out^* \wedge (\overline{x}, \overline{y}) \in \lfloor M \rfloor\}$$

Definition 10 *Stream X-machine M is* completely specified *if and only if $dom\, M = In^*$.*

It is straightforward to show that the stream X-machine given in Figure 2 is completely specified.

Where M is not completely specified, it is possible to complete $\lfloor M \rfloor$, to give $\lfloor M \rfloor_\perp$, using a symbol $\perp \notin In$ that represents the behaviour terminating with an error. $\lfloor M \rfloor_\perp$ is defined by the following [16].

Definition 11 *Given input sequence $\overline{x}$ and output sequence $\overline{y}$, $(\overline{x}, \overline{y}) \in \lfloor M \rfloor_\perp$ if and only if one of the following hold:*

(1) $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$.
(2) $\overline{x} \notin dom\, M$, $\overline{x} = \overline{x}_1 \overline{x}_2$ *for some maximal length* $\overline{x}_1 \in dom\, M$, $\overline{y} = \overline{y}_1 \perp$, *and* $(\overline{x}_1, \overline{y}_1) \in \lfloor M \rfloor$.

The first rule deals with the case where M is defined on $\overline{x}$ and the second rule deals with the case where M is not defined on $\overline{x}$. The second rule essentially says that the output sequence is found by following the sequence of outputs produced in response to the input sequence until a failure occurs. At this point the value $\perp$ is produced and no more output is observed.

Throughout this paper I will denote the implementation under test. As is usual, it will be assumed that the input and output domains of I are the same as those of the specification. Thus, since it will be assumed that I is deterministic, I is a function from the set of input sequences to the set of output sequences. Thus I has type $In^* \rightarrow Out^*$.

There are certain classes of stream X-machines.

Definition 12 *Stream X-machine $M = (In, Out, S, Mem, \Phi, F, s_0, m_0, \Gamma)$ is* deterministic *if and only if $\lfloor M \rfloor$ is a (possibly partial) function.*

Thus, if stream X-machine M is deterministic, for each input sequence $\overline{x} \in In^*$ there is at most one output sequence $\overline{y} \in Out^*$ such that $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$.

A number of different structural properties of a stream X-machine may lead to non-determinism. It is possible to restrict the sources of non-determinism in the specification.

Definition 13 *Stream X-machine $M = (In, Out, S, Mem, \Phi, F, s_0, m_0, \Gamma)$ is* quasi-non-deterministic *[16] if for all $s \in S$ and $f, f' \in \Phi$, if $(s, f), (s, f') \in dom\, F$ and $f \neq f'$ then $dom\, f \cap dom\, f' = \varnothing$.*

This means that, given the state, memory and input, at most one relation may be triggered. However, non-determinism may still occur through the relations not being functions. This restriction is applied by Hierons and Harman [16]. It will transpire that by removing this restriction we significantly alter the test generation problem.

The IUT I is *equivalent* to a stream X-machine M if and only if I and M define the same relation between input sequences and output sequences. This is the case if and only if $I = \lfloor M \rfloor_\perp$. Equivalence is the standard notion of correctness used where the specification and implementation are both deterministic.

When the specification is non-deterministic the appropriate notion of correctness is often weaker than equivalence. The specification gives a range of allowed behaviours and the behaviours in the IUT must be drawn from this. This alternative notion of correctness is often called conformance.

The IUT I conforms to stream X-machine M if and only if every input/output sequence (or trace) of I is also a trace of M. The following formally defines what it means for I to conform to M.

Definition 14 I conforms *to* M *if and only if* $I \subseteq \lfloor M \rfloor_\perp$. I *conforming to* M *will be denoted* $I \preceq M$.

The following is an immediate consequence of the above definition.

Proposition 1 *Assuming* I *behaves like some (possibly unknown) stream X-machine* M_I *with the same input alphabet as* M, I *conforms to* M *if and only if* $\lfloor M_I \rfloor_\perp \subseteq \lfloor M \rfloor_\perp$.

4 Testing and design for test conditions

When testing against a formal specification it is normal to assume that the implementation I is functionally equivalent to some element of a fault domain that contains a set of models described using a particular formal language (see, for example, [25]). When testing from a stream X-machine the fault domain contains stream X-machines: it is assumed that the implementation behaves like some unknown stream X-machine M_I with the same input and output alphabets, memory, and initial memory as M. Since we assume that it is known that the IUT I is deterministic, M_I must be deterministic. Further restrictions, called design for test conditions, are placed on M and the fault model.

It is worth briefly explaining why it may often be assumed that the model M_I has the same memory (Mem) as M. Recall that the memory models the values that may be passed between components from Φ: it acts like a (possibly infinite) central store that may be accessed and updated by any element from Φ. Since each component from M_I is known to conform to a component

from Φ and the interfaces of these components are known, the components from M_I do not access or affect values outside of this central store. Thus, the memory/central store of M_I is contained within that of M. It is possible to assume that M_I has memory *Mem* since values in *Mem* that are not required by M_I have no influence on testing. It is also assumed that M and M_I are initialized with the same values for the memory.

The design for test conditions may be divided into two groups [16]: specify for test conditions that place restrictions on Φ; and test hypotheses that place restrictions on M_I. These conditions will be described in the following.

When testing, test input may be chosen from a special set [24]. This might also restrict the possible memory values met in testing. These notions, based on those described by Ipate and Holcombe [24], will now be defined.

Definition 15 *A test environment $\mathcal{TE}$ is some pair $(\mathcal{M}, U)$, where $\mathcal{M} \subseteq Mem$ and $U : \Phi \to \mathcal{P}(In)$; we write $U(f)$ as U_f.*

The design for test conditions will be defined in terms of $\mathcal{TE}$. Essentially $\mathcal{TE}$ will be used to restrict the test input used: only input values from U_f will be used to try to trigger f. This weakens the overall design for test conditions by considering only some subset of values; those specified in $\mathcal{TE}$. Naturally, in some cases $\mathcal{TE}$ will allow any input: $\mathcal{M} = Mem$ and for all $f \in \Phi$, $U_f = In$.

It will be important that, when testing using $\mathcal{TE}$, values outside $\mathcal{M}$ are not met: the result of applying f with an input from U_f, when M has memory in $\mathcal{M}$ must lead to M having a memory value from $\mathcal{M}$. This is guaranteed if Φ is closed with respect to $\mathcal{TE}$ [24].

Definition 16 Φ *is* closed with respect to $\mathcal{TE}$ *if* $m_0 \in \mathcal{M}$ *and for all* $f \in \Phi$, $x \in U_f$, $m \in \mathcal{M}$, $y \in Out$, *and* $m' \in Mem$, *if* $((m, x), (y, m')) \in f$ *then* $m' \in \mathcal{M}$.

The design for test conditions will now be described.

Informally, Φ is output distinguishable with respect to $\mathcal{TE}$ if when restricting testing to values allowed by $\mathcal{TE}$, the output determines which relation has been applied. That is, given any two different relations $f_1, f_2 \in \Phi$, a memory value $m \in \mathcal{M}$, and an input value $x \in U_{f_1} \cup U_{f_2}$, the two relations cannot lead to the same output value if given x when the memory is m. This property allows the tester to associate input/output behaviour with relations from Φ [16,21].

Definition 17 Φ *is* output distinguishable with respect to $\mathcal{TE}$ *if for all* $f_1, f_2 \in \Phi$ *such that* $f_1 \neq f_2$, *all* $x \in U_{f_1} \cup U_{f_2}$, *all* $y \in Out$, *and all* $m, m' \in \mathcal{M}$ *such that* $((m, x), (y, m')) \in f_1$, *there does not exist* $m'' \in \mathcal{M}$ *such that*

$((m, x), (y, m'')) \in f_2$.

Informally, Φ is observable with respect to $\mathcal{TE}$ if, when restricting testing to $\mathcal{TE}$, the output from a relation can be used to determine the new memory value after its application. Observability allows the tester to determine the expected memory value based on the input and the output observed [16]. Without this property, it is difficult for the tester to determine an appropriate next input since this will typically depend on the current memory value.

Definition 18 Φ *is* observable with respect to $\mathcal{TE}$ *if and only if* $\forall f \in \Phi, m \in \mathcal{M}, x \in U_f$

$$(y_1, m_1), (y_2, m_2) \in f(m, x) \Rightarrow ((y_1 = y_2) \Rightarrow (m_1 = m_2)).$$

Possible ways of weakening this condition will be discussed in Section 9.

Informally Φ is complete with respect to $\mathcal{TE}$ if for each $f \in \Phi$, the tester can always apply an input from U_f, that is capable of triggering f, as long as the current memory value is known and is from $\mathcal{M}$. Note that this does not require that there actually be a transition from every state labelled with f, just that if there is such a transition then it can be followed by issuing an input from U_f regardless of memory.

Definition 19 Φ *is* complete with respect to $\mathcal{TE}$ *if* $\forall\, m \in \mathcal{M},\ f \in \Phi.\, \exists\, x \in U_f.(m, x) \in dom f$.

The following are the specify for test conditions. It is worth noting that they are weaker than those used by Hierons and Harman [16].

Definition 20 *If* Φ *is the relation set of a non-deterministic stream X-machine* $M = (In, Out, S, Mem, \Phi, F, s_0, m_0, \Gamma)$*, for which* $A(M)$ *is deterministic, and the test environment is* $\mathcal{TE}$ *then the* specify for test conditions *are:*

(1) Φ *is closed with respect to* $\mathcal{TE}$*;*
(2) Φ *is output distinguishable with respect to* $\mathcal{TE}$*;*
(3) Φ *is observable with respect to* $\mathcal{TE}$*;*
(4) Φ *is complete with respect to* $\mathcal{TE}$*.*

These conditions differ from those used by Hierons and Harman [16] only in the introduction of the test environment $\mathcal{TE}$. If $\mathcal{TE}$ allows all memory and input values, the specify for test conditions reduce to those previously given. However, as long as $\mathcal{TE}$ is closed with respect to Φ, reducing the set of values allowed by $\mathcal{TE}$ weakens the specify for test conditions applied to M. Naturally, they introduce conditions on $\mathcal{TE}$: not all choices of $\mathcal{TE}$ allow these specify for test conditions to be satisfied.

It has been noted that a stream X-machine which does not satisfy the specify for test conditions can always be rewritten to one that does satisfy these conditions [21]. This rewriting might involve the addition of new input and output values. Potentially these could either be removed or hidden when the system is released.

Consider the example given in Figure 2. In this paper we will use the test environment $\mathcal{T}E = (Mem, In)$: we will not restrict the input values that can be used in testing. The presence of the lights ensures that the operations are pairwise output distinguishable. The error operations guarantee that the specification is completely specified.

Since the test environment allows any memory value from Mem, Φ is immediately closed with respect to $\mathcal{T}E$: an operation cannot lead to a memory value outside the set given in $\mathcal{T}E$ since this set contains all the possible memory values. The print operation, PRINT and the six 'error' operations, SERR, DERR, PERR, RERR, NAERR and NRERR do not change the value of Mem and so these are vacuously observable. Since all the relations are actually functions, they are automatically observable.

To be complete with respect to $\mathcal{T}E$, every operation must have some input which triggers it in every memory from Mem. This can easily be verified. Thus, the example in Figure 2 satisfies the specify for test conditions.

The test hypotheses will now be described. It will be assumed that I behaves like some unknown stream X-machine $M_I = (In, Out, S', Mem, \Phi', F', s'_0, m_0, \Gamma')$. When testing from a deterministic stream X-machine it is normal to assume that M and M_I have the same sets of functions: faults may only occur through an incorrect state structure [21]. This assumption relates to either reusing trusted components or building a system from components that have been thoroughly tested. When testing for conformance, rather than equivalence, this assumption is relaxed to the assumption that each element of the set Φ' of relations of M_I conforms to some relation in M. A relation f' conforms to a relation f if and only if f' and f have the same preconditions and every pair in f' is also contained in f. A relation $f' \in \Phi'$ conforming to a relation $f \in \Phi$ will be denoted $f' \leq f$.

Definition 21 *Given $f' \in \Phi'$ and $f \in \Phi$, $f' \leq f$ if and only if $dom\, f' = dom\, f$ and $f' \subseteq f$. Further, $\Phi' \leq \Phi$ if and only if $\forall f' \in \Phi' \exists f \in \Phi . f' \leq f$.*

Informally, this means that f' conforms to f if they have the same input domain and any behaviour allowed by f' is also allowed by f. It is possible to extend $\leq$ to take sequences of relations, giving $\leq^*$ [16].

Suppose M_I has a relation set Φ' with $\Phi' \leq \Phi$. In a slight abuse of notation, it is possible to talk about Φ' satisfying the specify for test conditions with

$\mathcal{T}E$: for a relation $f' \in \Phi'$ $U_{f'} = U_f$ for the (unique[1]) relation $f \in \Phi$ with $f' \leq f$. Interestingly, if M_I has a relation set Φ' with $\Phi' \leq \Phi$, if M satisfies the specify for test conditions then M_I must also satisfy some of these. The following result is an immediate consequence of the definitions.

Proposition 2 *Suppose stream X-machine M, with relation set Φ, satisfies the specify for test conditions. If relation set $\Phi' \leq \Phi$ then:*

(1) Φ' is closed with respect to $\mathcal{T}E$;
(2) Φ' is observable with respect to $\mathcal{T}E$;
(3) Φ' is complete with respect to $\mathcal{T}E$.

It is now possible to formally state the two test hypotheses.

Definition 22 *If $M = (In, Out, S, Mem, \Phi, F, s_0, m_0, \Gamma)$ is a non-deterministic stream X-machine and I is the deterministic implementation to be tested against M then the* test hypotheses *are:*

(1) I behaves like some (unknown) minimal deterministic stream X-machine $M_I = (In, Out, S', Mem, \Phi', F', s'_0, m_0, \Gamma')$, for which $A(M_I)$ is deterministic, such that $\Phi' \leq \Phi$.
(2) There is some known n' such that M_I has at most n' states.

The design for test conditions given by Hierons and Harman [16] are a generalisation of those traditionally used when testing against deterministic stream X-machines. Thus these two test hypotheses together with the specify for test conditions are a generalisation of those traditionally used with deterministic stream X-machines.

It is often assumed that M is completely specified [24] and throughout the rest of the paper this assumption will be made. Where M is not completely specified, it may be converted into a completely specified stream X-machine by adding an error state and error messages. In order to maintain output distinguishability it may be necessary to use more than one error message. It will also be assumed that, for each input sequence, I has some corresponding behaviour and thus that M_I is completely specified. Section 9 will consider how these restrictions might be relaxed.

5 Characterising conformance

This section will characterise what it means for I to conform to M in terms of a relationship between the associated automata $A(M_I)$, the abstraction of the

[1] The uniqueness of f will be proved in Lemma 6.

implementation automaton, and $A(M)$, the abstraction of the specification. An algorithm that generates a test, that determines whether this relationship holds, will be given in Section 8.

Before developing the characterisation, those already considered in the literature will be described. For deterministic stream X-machines the characterisation is simple: I conforms to M if and only if $A(M)$ and $A(M_I)$ are equivalent [21]. Recent work has, however, considered the problem of testing against a non-deterministic stream X-machine.

It has been proved that testing to determine whether an implementation is equivalent to a non-deterministic stream X-machine may again be seen as a process of determining whether $A(M)$ and $A(M_I)$ are equivalent [24]. However, this is not the case when testing to determine whether an implementation I conforms to a quasi-non-deterministic stream X-machine M [16] since $A(M)$ and $A(M_I)$ could have different alphabets. To be precise, the relation set Φ' of M_I forms the alphabet of the automaton $A(M_I)$ and this need not be the same as the relation set Φ of M which forms the alphabet of the automaton $A(M)$.

A consequence of the design for test conditions (Lemma 6 below) is that for every $f' \in \Phi'$ there is exactly one $f \in \Phi$ such that $f' \leq f$. This relation f will be denoted $abs_\Phi(f')$. When comparing sequences of labels from $A(M)$ and $A(M_I)$, it is useful to introduce the abstraction, $Abs(M_I)$, of $A(M_I)$ formed by replacing each relation $f' \in \Phi'$ of M_I by the unique relation $abs_\Phi(f) \in \Phi$. Then, when M is quasi-non-deterministic, I conforms to M if and only if $Abs(M_I)$ is equivalent to $A(M)$ [16]. $Abs(M_I)$ may be formally defined in the following way.

Definition 23 *Given stream X-machine* $M_I = (In, Out, S', Mem, \Phi', F', s'_0, m_0, \Gamma')$ *and relation set* Φ *such that* $\Phi' \leq \Phi$, $Abs(M_I)$ *is the automaton* $(S', s'_0, \Phi, F'', \Gamma')$ *in which the function* F'' *is defined by the following.*

$$F'' = \{((s'_i, abs_\Phi(f')), s'_j) \mid ((s'_i, f'), s'_j) \in F'\}$$

Note that while Abs is parameterised by Φ, this parameter will remain implicit.

The situation considered in this paper is quite different from that considered previously. This is because I may conform to M even if $A(M)$ and $Abs(M_I)$ have very different structures. For example, if relations f_1 and f_2 leave a state s of M and $\mathrm{dom}\, f_1 = \mathrm{dom}\, f_2$, then it is possible that I conforms to M and yet M_I has only one relation f' leaving a corresponding state ($f' \leq f_1$ or $f' \leq f_2$). This is illustrated by the deterministic stream X-machine in Figure 3 that conforms to the stream X-machine given in Figure 2 but has a different structure; the slow division operation is removed, so that the division operation selected by

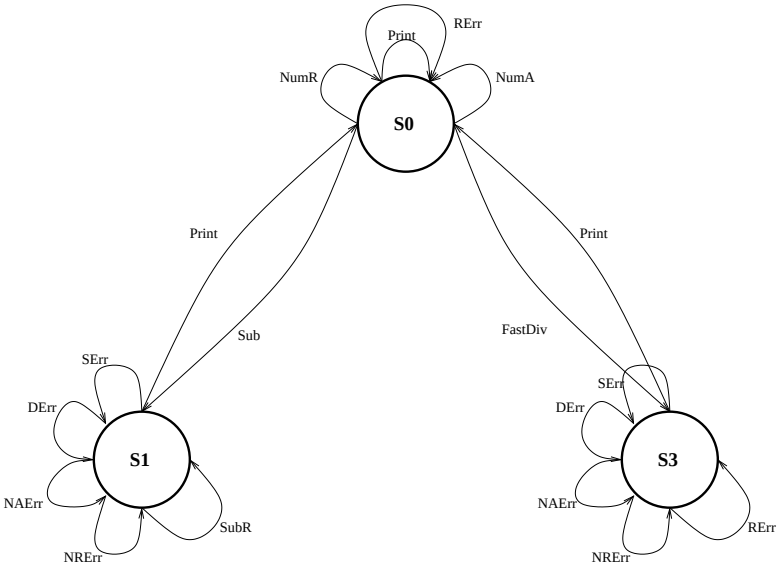

Fig. 3. A Correct Implementation

the $\mathcal{D}$ button is always the fast division operation. This situation cannot occur either when M is quasi-non-deterministic or when correctness is considered to be equivalence rather than conformance.

To further demonstrate how M and M_I may have different structures even if M_I conforms to M, consider the following class of examples. Given a set Φ of processing relations M_Φ is the chaos machine with one state s_0 and in which, for all $f \in \Phi$, there is a transition from s_0 to s_0 with label f. Assuming M_Φ is completely specified, *any* completely specified stream X-machine with relation set Φ', with $\Phi' \leq \Phi$, conforms to M_Φ. The restrictions applied in previous work did not allow such situations to occur.

Given that M_I may conform to M and yet have a radically different structure, the first challenge is to determine how M_I and M must relate in order for I to conform to M. Before stating this relationship, the notion of triggering a sequence $\overline{f} \in \Phi^*$ in a manner that is consistent with the test environment $\mathcal{T}E$, will be defined and some results will be proved. Essentially, an input/output sequence $\overline{x}/\overline{y}$ triggers $\overline{f} \in \Phi^*$ in a manner that is consistent with $\mathcal{T}E$ if each input is contained within the appropriate U_{f_i} and $\overline{x}/\overline{y}$ is contained in the relation of type $In^* \leftrightarrow Out^*$ defined by $\overline{f}$.

Definition 24 *Input/output sequence* $\overline{x}/\overline{y} = x_1, \ldots, x_k/y_1, \ldots, y_k$ *is* consistent with $\mathcal{T}E$ for $\overline{f} = f_1, \ldots, f_k \in \Phi^*$ *if there exists* $m_1, \ldots, m_k \in \mathcal{M}$ *such that, for all* $1 \leq i \leq k$*, the following hold*

(1) $x_i \in U_{f_i}$
(2) $((m_{i-1}, x_i), (y_i, m_i)) \in f_i$

Note that this means that if $\overline{x}/\overline{y}$ is consistent with the test environment $\mathcal{T}E$ for $\overline{f}$ then $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$.

We will now give some preliminary results which will be used, in Theorem 7, to define how $Abs(M_I)$ and $A(M)$ must relate in order for I to conform to M.

The following shows that every sequence from Φ^* has some input/output sequence that is consistent with $\mathcal{T}E$. This property allows testing to be restricted to using values from $\mathcal{T}E$.

Lemma 3 *Suppose the design for test conditions hold. Then given $\overline{f} \in \Phi^*$ there is some input/output sequence $\overline{x}/\overline{y}$ that is consistent with $\mathcal{T}E$ for $\overline{f}$.*

Proof

This follows using proof by induction on the length of $\overline{f}$ and from Φ being closed and complete with respect to $\mathcal{T}E$. □

The following shows that given a sequence of relations, that conforms to $\overline{f}$, it is possible to execute this sequence using values from $\mathcal{T}E$.

Lemma 4 *Suppose the design for test conditions hold. Then given $\overline{f} \in \Phi^*$ and $\overline{f}' \in \Phi'^*$, with $\overline{f}' \leq^* \overline{f}$, there is some input/output sequence $\overline{x}/\overline{y}$ that is consistent with the test environment $\mathcal{T}E$ for $\overline{f}$ such that $(\overline{x}, \overline{y}) \in \langle \overline{f}' \rangle$.*

Proof

Proof by induction on the length of $\overline{f}$. Clearly the result holds for the base case, the empty sequence.

Suppose the results hold for all sequences from Φ^* with length less than k ($k \geq 1$) and suppose $\overline{f}$ has length k. Then $\overline{f} = \overline{f}_1 f_2$ and $\overline{f}' = \overline{f}'_1 f'_2$, where $\overline{f}'_1 \leq^* \overline{f}_1$ and $f'_2 \leq f_2$. By the inductive hypothesis, there exist some input/output sequence $\overline{x}_1/\overline{y}_1$ that is consistent with $\mathcal{T}E$ for $\overline{f}_1$ such that $(\overline{x}_1, \overline{y}_1) \in \langle \overline{f}'_1 \rangle$. Let m denote the memory after $\overline{f}'_1$ is triggered with input $\overline{x}_1$ to produce output $\overline{y}_1$. By Proposition 2, Φ' is observable with respect to $\mathcal{T}E$ and so m is uniquely defined. By the definition of $\leq$ and the observability of Φ, m is also the memory after $\overline{f}_1$ is triggered with input $\overline{x}_1$ to produce output $\overline{y}_1$.

Since Φ is closed with respect to $\mathcal{T}E$, $m \in \mathcal{M}$. Observe that since Φ is complete with respect to $\mathcal{T}E$, there exists $x_2 \in U_{f_2}$ such that $(m, x_2) \in \operatorname{dom} f_2$. Suppose f'_2 responds to x_2 with output y_2 when in memory m. Then $\overline{x}_1 x_2/\overline{y}_1 y_2$ is consistent with $\mathcal{T}E$ for $\overline{f}$ and $(\overline{x}_1 x_2, \overline{y}_1 y_2) \in \langle \overline{f}' \rangle$. The result thus follows. □

Lemma 5 *Suppose the design for test conditions hold. Suppose also that $\overline{f}, \overline{g}$ are non-empty sequences from Φ^* such that there exists $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle \cap \langle \overline{g} \rangle$ that*

is consistent with the test environment $\mathcal{T}E$ for $\overline{f}$. Then $\overline{f} = \overline{g}$.

Proof

Proof by induction on the length of $\overline{f}$. Clearly the result holds for the base case, the empty sequence.

Suppose the results hold for all sequences from Φ^* with length less than k ($k \geq 1$) and suppose $\overline{f}$ has length k. Then $\overline{f} = \overline{f}_1 f$ and $\overline{g} = \overline{g}_1 g$, for some $\overline{f}_1, \overline{g}_1 \in \Phi^*$ and $f, g \in \Phi$. Further, $\overline{x} = \overline{x}_1 x$ and $\overline{y} = \overline{y}_1 y$ for some $\overline{x}_1 \in X^*$, $x \in X$, $\overline{y}_1 \in Y^*$, and $y \in Y$. Clearly $(\overline{x}_1, \overline{y}_1)$ is consistent with $\mathcal{T}E$ for $\overline{f}_1$. Thus, by the inductive hypothesis, $\overline{f}_1 = \overline{g}_1$.

Let m denote the unique memory value such that $((m_0, \overline{x}_1), (\overline{y}_1, m)) \in \|\overline{f}_1\|$. Then $((m, x), (y, m')) \in f$ and $((m, x), (y, m'')) \in g$ for some $m', m'' \in Mem$. Since $(\overline{x}, \overline{y})$ is consistent with $\mathcal{T}E$ for $\overline{f}$, and Φ is closed with respect to $\mathcal{T}E$, $m \in \mathcal{M}$. Further, since $(\overline{x}, \overline{y})$ is consistent with $\mathcal{T}E$ for $\overline{f}$, $x \in U_f$. The result now follows by observing that, since Φ is output distinguishable with respect to $\mathcal{T}E$, $f = g$. □

The following shows that abs_Φ and thus $Abs(M_I)$ is uniquely defined.

Lemma 6 *Suppose the design for test conditions hold. If $\overline{f}' \in \Phi'^*$, then there is exactly one sequence $\overline{f}$ in Φ^* with $\overline{f}' \leq^* \overline{f}$.*

Proof

This follows from Lemmas 4 and 5. □

The following states how M and M_I must relate for I to conform to M.

Theorem 7 *Suppose M is a stream X-machine that satisfies the specify for test conditions and I behaves like some deterministic stream X-machine M_I that satisfies the test hypotheses. I conforms to M if and only if the following conditions hold:*

(1) $L(Abs(M_I)) \subseteq L(A(M))$
(2) $dom\, M = dom\, M_I$

Proof

Case 1: ⇒

Suppose I conforms to M. By definition, $\mathrm{dom}\, M = \mathrm{dom}\, M_I$. Thus it is sufficient to prove that $L(Abs(M_I)) \subseteq L(A(M))$. Proof by contradiction will be used: suppose there exists $\overline{f} \in L(Abs(M_I)) \setminus L(A(M))$. Thus there is some $\overline{f}' \in L(A(M_I))$ such that $\overline{f}' \leq^* \overline{f}$.

By Lemma 4 there is some $(\overline{x}, \overline{y}) \in \langle \overline{f}' \rangle$ that is consistent with $\mathcal{T}E$ for $\overline{f}$. Since $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor$ and I conforms to M, $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$. Thus, there exists $\overline{f}_0 \in L(A(M))$ with $(\overline{x}, \overline{y}) \in \langle \overline{f}_0 \rangle$. Thus $(\overline{x}, \overline{y}) \in \langle \overline{f}_0 \rangle$ and $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$ and so, by Lemma 5, $\overline{f}_0 = \overline{f}$. Thus $\overline{f} \in L(A(M))$, providing a contradiction as required.

Case 2: $\Leftarrow$

Proof by contradiction: suppose conditions 1 and 2 hold but I does not conform to M. Then there exists minimal length $\overline{x} \in In^*$ and some sequence $\overline{y}$, of outputs possibly followed by $\perp$, such that $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor_\perp$ and $(\overline{x}, \overline{y}) \notin \lfloor M \rfloor_\perp$. Since dom M = dom M_I and $\overline{x}$ is minimal, $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor \setminus \lfloor M \rfloor$. Now consider sequence $\overline{f}' \in L(A(M_I))$ such that $(\overline{x}, \overline{y}) \in \langle \overline{f}' \rangle$. By condition 1 there is some $\overline{f} \in L(A(M))$ such that $\overline{f}' \leq^* \overline{f}$. Thus, since $(\overline{x}, \overline{y}) \in \langle \overline{f}' \rangle$ and $\overline{f}' \leq^* \overline{f}$, $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$. From this it follows that $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$, providing a contradiction as required. □

Since M and M_I are completely specified, the second condition is automatic.

Corollary 8 *If M is a completely specified stream X-machine that satisfies the specify for test conditions, and I behaves like some completely specified deterministic stream X-machine M_I that satisfies the test hypotheses, I conforms to M if and only if $L(Abs(M_I)) \subseteq L(A(M))$.*

Proof

Since M_I is completely specified dom $M_I = In^*$. The result thus follows from Theorem 7. □

The verification problem is now expressed as that of deciding whether $L(Abs(M_I)) \subseteq L(A(M))$. In Section 8 we will show how a finite test may be used to decide this.

6 The test process

This section will define the test process, that takes some $\overline{f} \in \Phi^*$ and tests the black-box implementation to determine whether $\overline{f} \in L(Abs(M_I))$. The test process will thus be used to determine whether some set of sequences from Φ^* is contained in $L(Abs(M_I))$. Section 8.2 will consider the problem of deriving some set $\mathcal{T}$ such that $L(Abs(M_I)) \subseteq L(A(M))$ if and only if $\mathcal{T} \subseteq L(Abs(M_I))$. Once such a set $\mathcal{T}$ has been found, we may determine whether the IUT conforms to M by applying the test process to the IUT with each sequence from $\mathcal{T}$. This leads to the IUT being executed with a set of test

sequences, each test sequence corresponding to some element of $\mathcal{T}$.

As with the quasi-non-deterministic case [16] the test process is adaptive: the next input depends upon the previous output observed. It thus produces a pair containing an input sequence and the corresponding output sequence observed in testing. Essentially, given $\overline{f}$, a test process tries to find some $(\overline{x}, \overline{y})$ that is consistent with the test environment $\mathcal{T}E$ for $\overline{f}$. If such a $(\overline{x}, \overline{y})$ can be found, $\overline{f}$ must be contained in $L(Abs(M_I))$. Since there may be more than one acceptable input at some point, there can be more than one possible test process.

Definition 25 *A test process for a non-deterministic stream X-machine M, with test environment $\mathcal{T}E$, is a function t of type $\Phi^* \rightarrow In^* \times Out^*$ that satisfies the following conditions:*

(1) $t(\epsilon) = (\epsilon, \epsilon)$.
(2) Suppose $\overline{f} \in L(A(M))$, $t(\overline{f}) = (\overline{x}_1, \overline{y}_1)$, *and* $((m_0, \overline{x}_1), (\overline{y}_1, m')) \in \left\|\overline{f}\right\|$. *Then there is some* $x \in U_f$ *such that* $(m', x) \in dom f$, *and if I produces output y in response to the input of x after* $\overline{x}_1/\overline{y}_1$, *then* $t(\overline{f}f) = (\overline{x}_1 x, \overline{y}_1 y)$.
(3) Suppose $\overline{f} \in L(A(M))$ *and* $t(\overline{f}) = (\overline{x}_1, \overline{y}_1)$. *If* $\neg\, \exists\, m \in Mem.((m_0, \overline{x}_1), (\overline{y}_1, m')) \in \left\|\overline{f}\right\|$, $t(\overline{f}f) = (\overline{x}_1, \overline{y}_1)$.
(4) If $\overline{f} \notin L(A(M))$, $t(\overline{f}f) = t(\overline{f})$.

Throughout this paper we assume the existence of a test process t.

The first rule is the base case, stating that testing based on the empty sequence requires no input and produces no output. The second and third rules are recursive cases, explaining how the test for sequence $\overline{f}f$ ($\overline{f} \in \Phi^*$, $f \in \Phi$) may be defined in terms of $t(\overline{f})$. The second rule gives the case where some $\overline{f}' \leq^* \overline{f}$ has been triggered by $t(\overline{f})$: here the sequence is extended by some value from U_f that should trigger f. The third rule covers the case where $t(\overline{f})$ has triggered some other sequence $\overline{f}' \not\leq^* \overline{f}$. In this paper the test process will be used to decide membership of $L(Abs(M_I))$, the language defined by the abstraction of the implementation machine, and thus, since at this point it has been determined that $\overline{f}$ is not contained in $L(Abs(M_I))$ the test need not be extended. The final rule states how a sequence $\overline{g} \in \Phi^*$ may be pruned, based on the observation that if there is some initial subsequence $\overline{f}$ of $\overline{g}$ such that $\overline{f} \notin L(A(M))$ then it is not necessary for the test process to test beyond $\overline{f}$: it is sufficient to decide whether $\overline{f} \in L(Abs(M_I))$. Note that I is an implicit parameter of the test process t.

Suppose the test process is applied to a sequence $\overline{f} = f_1, \ldots, f_k$ from the language $L(A(M))$ defined by the specification. The test process follows a sequence of steps. At the ith step, the test process produces an input x_i from

U_{f_i} that can trigger f_i, given the current memory. The input x_i is sent to the IUT I and the output is observed. From this, the memory after the transition may be determined.

The test process is not a function from Φ^* to input sequences: the next input used depends upon the output received in response to previous input. This is due to non-determinism in M and the fact that the next input will typically depend upon the memory value that has resulted from the previous behaviour. This memory value may be determined from the input/output behaviour since Φ is observable with respect to $\mathcal{T}E$.

The following results explain how the test process may be used to explore the relationship between $L(Abs(M_I))$, the language defined by the abstraction of the implementation, and $L(A(M))$, the language defined by the specification.

Lemma 9 *Suppose M and M_I satisfy the design for test conditions, t is a test process, $\overline{f} \in \Phi^*$ and $(\overline{x}, \overline{y}) = t(\overline{f})$. If $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$ then the sequence $\overline{f}' \in L(A(M_I))$ with $(\overline{x}, \overline{y}) \in \langle \overline{f}' \rangle$ satisfies $\overline{f}' \leq^* \overline{f}$.*

Proof

By the definition of a test process, $\overline{x}/\overline{y}$ is consistent with $\mathcal{T}E$ for $\overline{f}$. Consider the unique sequence $\overline{f}_1 \in \Phi^*$ with $\overline{f}' \leq^* \overline{f}_1$. Then $(\overline{x}, \overline{y}) \in \langle \overline{f}_1 \rangle \cap \langle \overline{f} \rangle$. The result now follows from Lemma 5. □

Note that a consequence of this result is that, under the conditions specified, we know that $\overline{f} \in L(Abs(M_I))$. The following shows the converse.

Lemma 10 *Suppose M and M_I satisfy the design for test conditions, t is a test process, $\overline{f} \in \Phi^*$ and $(\overline{x}, \overline{y}) = t(\overline{f})$. If $(\overline{x}, \overline{y}) \notin \langle \overline{f} \rangle$ then $\overline{f} \notin L(Abs(M_I))$.*

Proof

It is sufficient to prove that $\overline{f} \in L(Abs(M_I)) \Rightarrow t(\overline{f}) \in \langle \overline{f} \rangle$. This will be proved by induction on the length of $\overline{f}$. The result clearly holds for the base case, ϵ.

Suppose the result holds for every sequence of length less than k, $k > 0$, $\overline{f}$ has length k, and $\overline{f} \in L(Abs(M_I))$. Then $\overline{f} = \overline{f}_1 f$ for some $f \in \Phi$, $\overline{f}_1 \in \Phi^*$. Let $\overline{x} = \overline{x}_1 x$ and $\overline{y} = \overline{y}_1 y$ for some $x \in In$, $y \in Out$.

Since $\overline{f} \in L(Abs(M_I))$, $\overline{f}_1 \in L(Abs(M_I))$. By the inductive hypothesis, $(\overline{x}_1, \overline{y}_1) \in \langle \overline{f}_1 \rangle$. Suppose that $\overline{f}_1$ leads to memory m when triggered from the initial memory m_0 with input $\overline{x}_1$ and producing output $\overline{y}_1$. Since Φ is output distinguishable with respect to $\mathcal{T}E$, the behaviour $\overline{x}/\overline{y}$ in M_I can only occur through some $\overline{f}'_1 \in L(A(M_I))$ with $\overline{f}'_1 \leq^* \overline{f}_1$. Since Φ is observable with respect to $\mathcal{T}E$ the memory of M_I is m after $\overline{f}'_1$ and thus is m after $\overline{x}_1/\overline{y}_1$. Since Φ is closed

with respect to $\mathcal{T}E$, $m \in \mathcal{M}$.

Now consider the input of x in M_I after $\overline{x}_1/\overline{y}_1$. By the definition of t, $x \in U_f$ and $(m, x) \in \mathrm{dom}\, f$. Since M_I is deterministic, $\overline{f}_1 f \in L(Abs(M_I))$, and Φ is observable with respect to $\mathcal{T}E$, the input of x in M_I after $\overline{x}_1/\overline{y}_1$ must trigger some $f' \leq f$, $f' \in \Phi'$, and so there exists $m' \in Mem$ such that $((m, x), (y, m')) \in f'$. Thus, $(\overline{x}, \overline{y}) \in \langle \overline{f}'_1 f' \rangle$. The result thus follows from observing that $\overline{f}'_1 f' \leq^* \overline{f}$. □

7 Reaching and Distinguishing States

This section will initially consider the problem of finding a sequence from Φ^* that reaches a state s of the specification M and that must be implemented in the model M_I of the IUT if M_I conforms to M. The situation considered in this paper makes these issues significantly different from those considered in previous work. It will then consider the problem of finding sequences from Φ^* that distinguish the states of $A(M)$. Both of these types of sequences will be useful in test generation.

Before considering the problems of reaching and distinguishing states of $A(M)$, the notion of a sequence $\overline{f}$ being implemented in M_I will be defined.

Definition 26 *A sequence* $\overline{f} \in \Phi^*$ *is* implemented from state s'_i of M_I *if* $\overline{f} \in L_{Abs(M_I)}(s'_i)$. *A sequence* $\overline{f} \in \Phi^*$ *is* implemented *in* M_I *if it is implemented from the initial state of* M_I.

7.1 Reaching states of M

Due to non-determinism, in some cases a sequence $\overline{f}$ from M need not be implemented in M_I even if I conforms to M. This may happen where the input domain of $\overline{f}$ intersects the input domain of other sequences from $L(A(M))$. However, given a state s of M, it may be possible to identify sequences that must be implemented from any state of M_I that corresponds to s if I conforms to M. These are the sequences in the set $LD_M(s)$ defined below.

Definition 27 *A sequence* $\overline{f} = f_1, \ldots, f_k \in L_{A(M)}(s)$ *is contained in* $LD_M(s)$ *if and only if for all* $m \in \mathcal{M}$ *and* $\overline{x} = x_1, \ldots, x_k$, $x_i \in U_{f_i}$ *for all* i, $1 \leq i \leq k$, *such that* $(m, \overline{x}) \in dom\, \overline{f}$ *the following holds*

$$(m, \overline{x}) \notin \bigcup_{\overline{g} \in (L_{A(M)}(s) \setminus \{\overline{f}\})} dom\, \overline{g}$$

A consequence of this definition is that for each memory, $m \in \mathcal{M}$, and input sequence $\overline{x}$ that could be used by a test process to try to trigger $\overline{f}$, $(m, \overline{x})$ is in the input domain of $\overline{f}$ only. Thus, if M_I conforms to M then the behaviour of M_I in response to $\overline{x}$, when it is in a state s' corresponding to s and has memory m, must be consistent with $\overline{f}$. Since the input sequence $\overline{x}$ uses values from the appropriate U_{f_i}, if the corresponding behaviour is seen in M_I then, due to output distinguishability with respect to $\mathcal{T}E$, it can only have arisen through the execution of some $\overline{f}'$ with $\overline{f}' \leq^* \overline{f}$. From this it is possible to deduce that $\overline{f} \in L_{Abs(M_I)}(s')$.

Interestingly, the above condition may be weakened: it is sufficient that for each $m \in \mathcal{M}$ there is some such input sequence $\overline{x}$. Such a definition might state that a sequence $\overline{f} = f_1, \ldots, f_k \in L_{A(M)}(s)$ is contained in $LD'_M(s)$ if and only if for all $m \in \mathcal{M}$ there exists $\overline{x} = x_1, \ldots, x_k$, $x_i \in U_{f_i}$ such that $(m, \overline{x}) \in \mathrm{dom}\, \overline{f}$ and the following holds.

$$(m, \overline{x}) \notin \bigcup_{\overline{g} \in (L_{A(M)}(s) \setminus \{\overline{f}\})} \mathrm{dom}\, \overline{g}$$

However, if the weaker condition, based on LD'_M, is used then the test process must be defined in a more complex manner in order to ensure that it uses the appropriate input sequence where we are relying on a sequence being contained in $LD'_M(s)$. The above definition (Definition 27) of $LD_M(s)$ will be used throughout this paper in order to aid readability.

The following shows that sequences from $LD_M(s)$ must be implemented in M_I if M_I conforms to M on certain sequences.

Lemma 11 *Let M and M_I satisfy the design for test conditions. Let $Abs(M_I)$ have initial state s'_0 and next state function F'. Then for all $\overline{f}_1 \in L(A(M)) \cap L(Abs(M_I))$, if $F^*(s_0, \overline{f}_1) = s$, $F'^*(s'_0, \overline{f}_1) = s'$, $\overline{f} \in LD_M(s)$, and $t(\overline{f}_1\overline{f}) \in \lfloor M \rfloor$ then we have that $\overline{f} \in L_{Abs(M_I)}(s')$.*

Proof

Suppose $(\overline{x}, \overline{y}) = t(\overline{f}_1\overline{f})$, $\overline{x} = \overline{x}_1\overline{x}_2$, $\overline{y} = \overline{y}_1\overline{y}_2$, and $|\ \overline{x}_1\ | = |\ \overline{y}_1\ | = |\ \overline{f}_1\ |$.

Since $\overline{f}_1 \in L(A(M)) \cap L(Abs(M_I))$, by Lemma 10, $(\overline{x}_1, \overline{y}_1) = t(\overline{f}_1) \in \langle \overline{f}_1 \rangle$. Suppose $m \in Mem$ has the property that $((m_0, \overline{x}_1), (\overline{y}_1, m)) \in \|\overline{f}_1\|$. Since Φ is closed with respect to $\mathcal{T}E$, $m \in \mathcal{M}$. Thus, since $\overline{f} \in LD_M(s)$, $F^*(s_0, \overline{f}_1) = s$, and $t(\overline{f}_1\overline{f}) \in \lfloor M \rfloor$, $t(\overline{f}_1\overline{f}) \in \langle \overline{f}_1\overline{f} \rangle$. Thus, by Lemma 9, $\overline{f}_1\overline{f} \in L(Abs(M_I))$. The result now follows. □

Definition 28 *A state of M reached by a sequence in $LD_M(s_0)$ is said to be* deterministically reachable *or* d-reachable. *A sequence $\overline{v} \in LD_M(s_0)$ with $F^*(s_0, \overline{v}) = s$ is said to* d-reach *s.*

Based on this definition, it is possible to define classes of sets that will be used in test generation.

Definition 29 *A set $V \subseteq LD_M(s_0)$ is a* deterministic state cover *if no state of M is reached by more than one sequence from V and V contains the empty sequence ϵ. Given V, the set $S_V \subseteq S$ will denote the set of states of M d-reached by sequences in V.*

Note that V and S_V are non-empty since s_0 is d-reached by ϵ.

Naturally, it is normally desirable that V contains sequences that d-reach all the d-reachable states in M but this restriction will not be introduced. Throughout this paper V will denote a deterministic state cover that has been chosen and will be used in testing.

Now consider the example. Both the FastDiv operation or the SlowDiv operation can be triggered by the $\mathcal{D}$ button. Further, these are the only transitions that reach states $S2$ and $S3$. Thus $S2$ and $S3$ are not deterministically reachable. Clearly $S0$ is deterministically reachable by ϵ and $S1$ is deterministically reachable by $<$ Sub $>$. We may choose the state cover, V, to be the set $\{\epsilon, <$ Sub $>\}$ which reach the deterministically reachable (or d-reachable) states. The set of d-reachable states, S_V reached by elements of V is $\{S0, S1\}$.

It is worth noting that the restriction placed on V, that it contains the empty sequence, is required. This is because it will transpire that every test will start with a sequence from V. If V does not contain the empty sequence then there may be some relation f' implemented from the initial state of M_I such that f' reaches erroneous parts of the implementation and no sequence starting with an element of V can reach these sections of M_I. Such classes of faults could not be detected by tests starting with V. Observe that a similar restriction on the state cover is made in the W-method [7].

7.2 Distinguishing states of $A(M)$

When generating tests from a state-based specification, it is important to decide how states of the implementation may be distinguished. This might be based on sequences that distinguish states of the specification. This section will consider the problem of distinguishing states of the specification automaton $A(M)$.

It is possible to generate sequences that distinguish states of $A(M)$ from state $s \in S$ by considering sequences in $LD_M(s)$, since these must be implemented in any state of M_I corresponding to s. Such a sequence, $\overline{f}$, distinguishes s from some state $s_i \in S$ if $\overline{f}$ does not label a path leaving s_i.

Definition 30 *A sequence* $\overline{f}$ distinguishes *states* s_i *and* s_j *of* $A(M)$ *if* $\overline{f} \in LD_M(s_i)$ *and* $\overline{f} \notin L_{A(M)}(s_j)$. *If* $\overline{f}$ *distinguishes* s_i *and* s_j *then* $\overline{f}$ *distinguishes* s_j *and* s_i. *If some sequence distinguishes* s_i *and* s_j *then* s_i *and* s_j *are said to be* distinguishable.

Sequences that distinguish states of $A(M)$ will be used in testing. The following shows their value: if $\overline{f}$ distinguishes two states s_1 and s_2 of $A(M)$ then $\overline{f}$ can be used to distinguish corresponding states of M_I.

Lemma 12 *Suppose the design for test conditions hold and states* s_1 *and* s_2 *of* $A(M)$ *are distinguished by* $\overline{f}$. *Suppose* $\overline{f}_1$ *and* $\overline{f}_2$ *reach states* s_1' *and* s_2' *respectively of* $Abs(M_I)$, $F^*(s_0, \overline{f}_1) = s_1$, *and* $F^*(s_0, \overline{f}_2) = s_2$. *If* $t(\overline{f}_1\overline{f})$ *and* $t(\overline{f}_2\overline{f})$ *are input/output sequences in the specification* $(t(\overline{f}_1\overline{f}), t(\overline{f}_2\overline{f}) \in \lfloor M \rfloor)$ *then* $\overline{f}$ *distinguishes states* s_1' *and* s_2' *of* $Abs(M_I)$.

Proof

Let $\overline{f} = f_1, \ldots, f_k$. Without loss of generality $\overline{f} \in L_{A(M)}(s_1) \setminus L_{A(M)}(s_2)$ and $\overline{f} \in LD_M(s_1)$. Since $\overline{f} \in LD_M(s_1)$, for all $m \in \mathcal{M}$ and $\overline{x} = x_1, \ldots, x_k$ such that $x_i \in U_{f_i}$ $(1 \leq i \leq k)$ and $(m, \overline{x}) \in \operatorname{dom} \overline{f}$

$$(m, \overline{x}) \notin \bigcup_{\overline{f}_1 \in (L_{A(M)}(s_1) \setminus \{\overline{f}\})} \operatorname{dom} \overline{f}_1$$

Observe that since $\overline{f}_1, \overline{f}_2 \in L(A(M)) \cap L(Abs(M_I))$, by Lemma 10, $t(\overline{f}_1) \in \langle \overline{f}_1 \rangle$ and $t(\overline{f}_2) \in \langle \overline{f}_2 \rangle$. Since $t(\overline{f}_1\overline{f}) \in \lfloor M \rfloor$ and $\overline{f} \in LD_M(s_1)$, $t(\overline{f}_1\overline{f}) \in \langle \overline{f}_1\overline{f} \rangle$.

Since $t(\overline{f}_1\overline{f}) \in \lfloor M \rfloor$, by Lemma 11, $\overline{f} \in L_{Abs(M_I)}(s_1')$.

Since $t(\overline{f}_2\overline{f}) \in \lfloor M \rfloor$ and $\overline{f}_2\overline{f} \notin L(A(M))$, $t(\overline{f}_2\overline{f}) \notin \langle \overline{f}_2\overline{f} \rangle$. Thus, by Lemma 10, $\overline{f}_2\overline{f} \notin L(Abs(M_I))$. But $\overline{f}_2 \in L(Abs(M_I))$. Thus, $\overline{f} \notin L_{Abs(M_I)}(s_2')$.

We thus have that $\overline{f} \in L_{Abs(M_I)}(s_1') \setminus L_{Abs(M_I)}(s_2')$. Since M_I is deterministic and Φ is observable with respect to $\mathcal{T}E$, $\overline{f} \in LD_{Abs(M_I)}(s_1')$ and thus the result follows. □

The following notation will be used in this paper.

Definition 31 *Given set* A *and* $\overline{d} \in A^*$, $Pre(\overline{d}) = \{\overline{d}_1 \mid \exists\, \overline{d}_2 \in A^*.\overline{d} = \overline{d}_1\overline{d}_2\}$ *denotes the set of initial subsequences of* $\overline{d}$. *Given* $D \subseteq A^*$, $Pre(D) = \{\overline{d} \mid \exists\, \overline{d}_1 \in D.\overline{d} \in Pre(\overline{d}_1)\}$.

A set $W \subseteq \Phi^*$ will be used to distinguish states of the implementation. The set W will be called a *characterizing set*. Ideally, the states in every pair (s_i, s_j) of distinguishable states of $A(M)$ are distinguished by some element of $Pre(W)$. However, this restriction will not be placed on W. It will be assumed that such a set W has been chosen and will be used in testing.

By Lemma 12, a sequence $\overline{f}$ that distinguishes two states of the specification must distinguish corresponding states of the implementation. Thus, if a characterizing set W distinguishes states of the specification then it may be used to distinguish between states of $Abs(M_I)$ during testing.

Now consider the example. The set of pairwise distinguishable states are as follows:

$<$ SUB $>$	distinguishes	$S0$ and $S1$
		$S0$ and $S2$
		$S0$ and $S3$
$<$ SUBR $>$	distinguishes	$S2$ and $S3$
$<$ SUBR $>$	distinguishes	$S1$ and $S3$

There is no sequence which distinguishes $S1$ and $S2$ since they are not pairwise distinguishable. This feature, along with $S2$ and $S3$ not being deterministically reachable, make this example interesting from the point of view of testing a deterministic implementation against a non-deterministic specification.

For each pairwise distinguishable pair of states, the set W should ideally contain a sequence which distinguishes them. In this case W could be the set $\{<$ SUB $>, <$ SUBR $>\}$, since SUB distinguishes $S0$ from $S1$, $S2$ and $S3$, while SUBR distinguishes $S2$ from $S3$ and $S1$ from $S3$. $S1$ and $S2$ are not pairwise distinguishable.

8 Test generation

The problem of determining whether I conforms to M has been shown to be equivalent to the problem of determining whether the language defined by the abstraction of the IUT, $L(Abs(M_I))$, is contained in the language $L(A(M))$ defined by the specification. In order to explore $L(Abs(M_I))$, tests will be produced and applied to the IUT in order to determine whether certain sequences from Φ^* are contained in $L(Abs(M_I))$. Section 8.1 will define the product machine $P(M, M_I)$ and represent the problem of determining whether I conforms to M as one of deciding whether the state *Fail* of $Abs(P(M, M_I))$ is reachable from its initial state. Section 8.2 uses an approach based on state counting [30,31,37] to produce a finite set $\mathcal{T} \subseteq \Phi^*$ with the property that the state *Fail* of $Abs(P(M, M_I))$ is reachable if and only if it is reached by some input/output sequence triggered by the application of the test process to some element of $\mathcal{T}$. Testing, by applying the test process to each element of $\mathcal{T}$, is thus guaranteed to determine correctness under the design for test conditions.

This section will describe the notion of the product machine $P(M, M_I)$, formed from M and M_I, that has a special state *Fail*. The definition of the product machine relates to a similar notion used in testing a deterministic implementation against a non-deterministic finite state machine [31]. Having defined the product machine, Lemma 13 will give a relationship between the sequences of $Abs(P(M, M_I))$ and those of $A(M)$ and $Abs(M_I)$. In Lemma 14 it will be proved that $L(Abs(M_I)) \subseteq L(A(M))$ if and only if the state *Fail* of $Abs(P(M, M_I))$ is not reachable. Finally, in Theorem 15, it will be proved that I conforms to M if and only if the state *Fail* of $Abs(P(M, M_I))$ is not reachable. The next section will consider the problem of testing to determine whether *Fail* is reachable.

The product machine is a stream X-machine with the same memory and input and output alphabets as M and M_I. It is related to the machine formed by executing M and M_I in parallel: the state of the product machine is either the states of M_I and M, corresponding to the behaviour observed in testing, or the state *Fail*. When an input is provided, the product machine finds the appropriate relation f' from M_I to trigger. If the current state of the product machine allows some transition from M with relation $f \in \Phi$ with $f' \leq f$, the transitions corresponding to f' and f are taken. Otherwise the product machine moves to the state *Fail*. Naturally, since M_I is unknown before testing, the product machine is also unknown before testing. However, the notion of the product machine will prove to be useful when reasoning about test effectiveness.

The product machine $P(M, M_I)$ will now be defined.

Definition 32 *The* product machine $P(M, M_I)$ *formed from* $M = (In, Out, S, Mem, \Phi, F, s_0, m_0, S)$ *and* $M_I = (In, Out, S', Mem, \Phi', F', s'_0, m_0, S')$*, is the stream X-machine* $(In, Out, S_P, Mem, \Phi', F_P, (s_0, s'_0), m_0, S_P)$ *in which* $S_P = (S \times S') \cup \{Fail\}$ *(*$Fail \notin S \times S'$*) and the (partial) next state function* F_P *is defined by the following rules.*

- *For all* $f' \in \Phi'$, $F_P(Fail, f') = Fail$.
- *Given state* $(s, s') \in S_P$ *and* $f' \in \Phi'$, $F_P((s, s'), f')$ *is defined by:*

(1) If $(s', f') \in dom\, F'$ *and there exists* $f \in \Phi$ *such that* $(s, f) \in dom\, F$ *and* $f' \leq f$ *then* $F_P((s, s'), f') = (F(s, f), F'(s', f'))$

(2) Else if $(s', f') \in dom\, F'$ *then* $F_P((s, s'), f') = Fail$.

In a slight abuse of notation, F_P with be used to denote the transition function for both $P(M, M_I)$ and $Abs(P(M, M_I))$. Similarly, F' will be used to denote the (partial) transition function for M_I, $A(M_I)$, and $Abs(M_I)$.

The following relates sequences in $A(P(M, M_I))$ to those in $A(M)$ and $A(M_I)$.

Lemma 13 *Suppose the design for test conditions hold and* $\overline{f}' \in \Phi'^*$, $\overline{f} \in \Phi^*$ *with* $\overline{f}' \leq^* \overline{f}$. *Then* $F_P^*((s_0, s_0'), \overline{f}') = (s, s')$ *if and only if* $F^*(s_0, \overline{f}) = s$ *and* $F'^*(s_0', \overline{f}') = s'$.

Proof

There are two cases to consider.

Case 1: $\Rightarrow$

Proof by induction on the length of $\overline{f}'$. The base case, in which $\overline{f}' = \epsilon$, clearly holds.

Suppose now that the result holds for all sequences in Φ'^* of length less than k, $k \geq 1$, and $\overline{f}'$ has length k. Then $\overline{f}' = \overline{f}_1' f_2'$ for some $f_2' \in \Phi'$ and $\overline{f} = \overline{f}_1 f_2$ for some $\overline{f}_1$ and f_2 with $\overline{f}_1' \leq^* \overline{f}_1$ and $f_2' \leq f_2$.

Suppose that $F_P^*((s_0, s_0'), \overline{f}') = (s, s')$ and consider the state $(s_i, s_j') = F_P((s_0, s_0'), \overline{f}_1')$. By the inductive hypothesis, $F^*(s_0, \overline{f}_1) = s_i$ and $F'^*(s_0', \overline{f}_1') = s_j'$. By the definition of F_P and the fact that $F_P^*((s_0, s_0'), \overline{f}') = (s, s')$, $F(s_i, f_2) = s$ and $F'(s_j', f_2') = s'$. Thus $F^*(s_0, \overline{f}) = s$ and $F'^*(s_0', \overline{f}') = s'$ as required.

Case 2: $\Leftarrow$

Proof by induction on the length of $\overline{f}'$. The base case, in which $\overline{f}' = \epsilon$, clearly holds.

Suppose now that the result holds for all sequences in Φ'^* of length less than k, $k \geq 1$, and $\overline{f}'$ has length k. Then $\overline{f}' = \overline{f}_1' f_2'$ for some $f_2' \in \Phi'$ and $\overline{f} = \overline{f}_1 f_2$ for some $\overline{f}_1$ and f_2 with $\overline{f}_1' \leq^* \overline{f}_1$ and $f_2' \leq f_2$.

Suppose that $F^*(s_0, \overline{f}) = s$ and $F'^*(s_0', \overline{f}') = s'$. Consider the states $s_i = F^*(s_0, \overline{f}_1)$ and $s_j' = F'^*(s_0', \overline{f}_1')$. Clearly $F(s_i, f_2) = s$ and $F'(s_j', f_2') = s'$.

By the inductive hypothesis, $F_P^*((s_0, s_0'), \overline{f}_1') = (s_i, s_j')$. By the definition of F_P, $F_P((s_i, s_j'), f_2') = (s, s')$. Thus $F_P^*((s_0, s_0'), \overline{f}') = (s, s')$ as required. $\square$

It is worth noting that, by Lemma 3, if state *Fail* of $Abs(P(M, M_I))$ is reached using $\overline{f}$, the state *Fail* of $P(M, M_I)$ can be reached using an input sequence consistent with $\mathcal{T}E$ for $\overline{f}$. The following relates the property of interest, $L(Abs(M_I)) \subseteq L(A(M))$, to the structure of the product machine.

Lemma 14 *Suppose the design for test conditions hold. Then* $L(Abs(M_I)) \subseteq L(A(M))$ *if and only if the state Fail of* $Abs(P(M, M_I))$ *cannot be reached*

from the initial state of $Abs(P(M, M_I))$.

Proof

Case 1: $\Rightarrow$

Proof by contradiction: suppose $L(Abs(M_I)) \subseteq L(A(M))$ and the state *Fail* of $Abs(P(M, M_I))$ is reachable. Consider a minimal length sequence $\overline{f}$ that reaches state *Fail* of $Abs(P(M, M_I))$. By the definition of F_P and the minimality of $\overline{f}$, $\overline{f} \in L(Abs(M_I))$. Suppose $\overline{f}' \leq^* \overline{f}$, $\overline{f}' \in L(A(M_I))$. Let $\overline{f}' = \overline{f}'_1 f'_2$ for some $f'_2 \in \Phi'$. Suppose $\overline{f}'_1 \leq^* \overline{f}_1$, $f'_2 \leq f_2$ ($f_2 \in \Phi$, $\overline{f}_1 \in \Phi^*$). Then by the minimality of $\overline{f}$, $F^*_P((s_0, s'_0), \overline{f}'_1) = (s, s')$ for some $s \in S$ and $s' \in S'$.

By Lemma 13, $s = F^*(s_0, \overline{f}_1)$ and $s' = F'^*(s'_0, \overline{f}'_1)$. Further, by the definition of F_P, since $F_P((s, s'), f'_2) = Fail$, $(s, f_2) \notin \operatorname{dom} F$. Thus, $\overline{f}_1 f_2 \notin L(A(M))$ and so $\overline{f} \notin L(A(M))$, contradicting $L(Abs(M_I)) \subseteq L(A(M))$ as required.

Case 2: $\Leftarrow$

Proof by contradiction: suppose the state *Fail* is not reachable but that $L(Abs(M_I)) \not\subseteq L(A(M))$. Let $\overline{f}$ be some minimal length element of $L(Abs(M_I)) \setminus L(M)$ and $\overline{f}' \leq^* \overline{f}$ for $\overline{f}' \in L(A(M_I))$. Since F^*_P is specified for all sequences in $L(A(M_I))$ and the state *Fail* is not reachable, $F^*_P((s_0, s'_0), \overline{f}') = (s, s')$ for some (s, s'). Thus, by Lemma 13, $F^*(s_0, \overline{f}) = s$ and so $\overline{f} \in L(A(M))$, providing a contradiction as required. □

The following expresses the problem of deciding correctness in terms of deciding state reachability in $Abs(P(M, M_I))$.

Theorem 15 *Suppose the design for test conditions hold. If* M *and* M_I *are completely specified then* M_I *conforms to* M *if and only if the state Fail of* $Abs(P(M, M_I))$ *cannot be reached from the initial state of* $Abs(P(M, M_I))$.

Proof

The result follows from Corollary 8 and Lemma 14. □

Testing may now be seen as a problem of determining whether the state *Fail* of $Abs(P(M, M_I))$ is reachable from (s_0, s'_0). The next section will consider the problem of applying black-box testing in order to decide this.

The following two results show that a sequence $\overline{f}$ reaches that state *Fail* of $Abs(P(M, M_I))$ if and only if testing using the test process, applied to $\overline{f}$, finds a failure.

Lemma 16 *Suppose the design for test conditions hold. If* $\overline{f} \in \Phi^*$ *reaches the*

state Fail of $Abs(P(M, M_I))$ then the behaviour produced by the test process applied to $\overline{f}$ and I is not allowed by the specification ($t(\overline{f}) \notin \lfloor M \rfloor$).

Proof

Let $\overline{g}$ denote a minimal initial subsequence of $\overline{f}$ that reaches *Fail* and $\overline{g} = \overline{g}_1 g$ ($g \in \Phi$). Thus $\overline{g}_1$ reaches some state (s, s') of the product machine and $\overline{g}_1 \in L(A(M)) \cap L(Abs(M_I))$.

If $t(\overline{g}_1) \notin \lfloor M \rfloor$, the result follows immediately. Suppose $t(\overline{g}_1) \in \lfloor M \rfloor$ and that the memory in M is $m \in \mathcal{M}$ after behaviour $t(\overline{g}_1) = (\overline{x}, \overline{y})$. By Lemma 5, since $\overline{g}_1 \in L(A(M))$, the behaviour $(\overline{x}, \overline{y})$ is achieved in M through $\overline{g}_1$. Similarly, since $\overline{g}_1 \in L(Abs(M_I))$ and Φ is output distinguishable with respect to $\mathcal{T}E$, the behaviour $(\overline{x}, \overline{y})$ in M_I is achieved through some $\overline{g}'_1 \leq^* \overline{g}_1$. Since Φ is observable with respect to $\mathcal{T}E$, the memory of M_I after $(\overline{x}, \overline{y})$ is m. The test process now applies some $x \in U_g$ such that $(m, x) \in \operatorname{dom} g$.

By the definition of $P(M, M_I)$, since $F_P((s, s'), g) = \mathit{Fail}$, $(s', g) \in \operatorname{dom} F'$ and $(s, g) \notin \operatorname{dom} F$. Suppose I produces output y after the input of x following the input/output sequence of $\overline{x}/\overline{y}$. Since, $(s, g) \notin \operatorname{dom} F$, by the output distinguishability of Φ with respect to $\mathcal{T}E$, M cannot allow output y after the input/output behaviour $\overline{x}/\overline{y}$. Thus, $(\overline{x}x, \overline{y}y) \notin \lfloor M \rfloor$ and so $t(\overline{f}) \notin \lfloor M \rfloor$ as required. □

Lemma 17 *Suppose the design for test conditions hold. If the behaviour produced by the test process applied to $\overline{f}$ and I is not allowed by the specification ($t(\overline{f}) \notin \lfloor M \rfloor$) then $\overline{f}$ reaches the state Fail of $Abs(P(M, M_I))$.*

Proof

Proof by contradiction: suppose that $t(\overline{f}) \notin \lfloor M \rfloor$ but that $\overline{f}$ does not reach the state *Fail* of $Abs(P(M, M_I))$. Then $\overline{f}$ must reach some state (s, s') of $Abs(P(M, M_I))$. By Lemma 13, $\overline{f} \in L(A(M)) \cap L(Abs(M_I))$. By Lemma 10, $t(\overline{f}) \in \langle \overline{f} \rangle$. This contradicts $t(\overline{f}) \notin \lfloor M \rfloor$ as required. □

8.2 Generating tests

This section will consider the problem of finding some set $\mathcal{T}$ of sequences from Φ^* such that the state *Fail* of $Abs(P(M, M_I))$ is reachable if and only if the test process applied to the IUT with $\overline{\tau}$ leads to a failure for some $\overline{\tau} \in \mathcal{T}$. By Lemmas 16 and 17, if the product machine was known it would be sufficient to produce a set of tests that, between them, reach every state of $Abs(P(M, M_I))$. While the product machine is not known, it is possible to reason about it and, on this basis, to limit testing. In particular, testing can be seen as a process of

searching for a *minimal length* sequence to the state *Fail*. Thus, where it can be shown that a sequence leads to a repeated state of $Abs(P(M, M_I))$, this sequence need not be extended.

Lemma 18 will give a sufficient condition for a sequence from Φ^* to meet some state of $Abs(P(M, M_I))$ that has already been met. This condition will be used, in test generation, to determine when an element of Φ^* need not be extended further. It thus forms the basis of test generation: keep on extending sequences until the condition is satisfied. The condition is based on the idea of state counting which has previously been used in testing from a non-deterministic finite state machine [31].

Lemma 19 shows that the proposed test is finite irrespective of the choice of V and W. In Theorem 21 it is proved that if I passes the test then the state *Fail* of $Abs(P(M, M_I))$ is not reachable. Lemma 22 then proves the converse: that if state *Fail* of $Abs(P(M, M_I))$ is not reachable then I passes the test. These results are summarised in Theorem 23. Finally, in Theorem 24, results are brought together to prove that the test determines correctness under the design for test conditions.

8.2.1 Initial observations

Recall that n' is an upper bound on the number of states of M_I. The first observation that may be made is that, since $Abs(P(M, M_I))$ has at most $n'n+1$ reachable states, the state *Fail* is reachable if and only if it is reachable by some sequence of length no more than $n'n$. Thus it is sufficient to use the set $\mathcal{T}_{n'n} = \Phi^{n'n}$ and test by applying the test process to the IUT using every element of $\mathcal{T}_{n'n}$. However, it will transpire that a smaller test will suffice.

8.2.2 Exploring the product machine

This section will adapt the notion of state counting [30,31,37], to reason about the states of the product machine reached during testing. Throughout this section V and W will denote the deterministic state cover and the characterising set to be used in testing. Given set $K \subseteq S$, $\hat{K}$ will denote the set of states of K that are reached by sequences in V: $\hat{K} = K \cap S_V$. Given $\overline{v} \in V$, $s_{\overline{v}}$ denotes the state $F^*(s_0, \overline{v})$.

The set W pairwise distinguishes the states of some $T \subseteq S$ if and only if for every $s, s' \in T$ with $s \neq s'$, some sequence from W distinguishes s and s'. The following notation will be used in state counting.

Definition 33 *Suppose that the set T of states of M are pairwise distinguished by W. Let $n(s, \overline{v}, \overline{f})$ denote the number of times that s is met by follow-*

ing $\overline{f}$ *from* $s_{\overline{v}}$ *in* $A(M)$. *Thus* $n(s, \overline{v}, \overline{f}) =\mid \{\overline{g} \in Pre(\overline{f}) \setminus \{\epsilon\} \mid F^*(s_{\overline{v}}, \overline{g}) = s\} \mid$. *Then*

$$n(T, \overline{v}, \overline{f}) = \sum_{s \in T} n(s, \overline{v}, \overline{f})$$

Given sets A and B of sequences, AB will denote $\{\bar{c} \mid \exists\, \bar{a} \in A, \bar{b} \in B.\bar{c} = \bar{a}\bar{b}\}$. Suppose that it is known, from testing, that $VW \subseteq L(Abs(M_I))$ (I has been tested by applying the test process to each element of VW and no failures have been observed). Consider now a sequence in the form of $\overline{v}\overline{f}$ for $\overline{v} \in V$ and $\overline{f} \in \Phi^*$. Suppose testing demonstrates that the set $Pre(\overline{v}\overline{f})\,W$ is contained in $L(Abs(M_I))$. It is now possible to consider the states of $Abs(P(M, M_I))$ met by $\overline{v}\overline{f}$ and V. The following gives a sufficient condition for there to have been a repetition in the states visited and thus a condition under which a sequence $\overline{v}\overline{f}$ need not be extended.

Lemma 18 *Suppose the design for test conditions hold,* $\overline{v}\overline{f} \in L(A(M))$, *and for all* $\overline{\tau} \in VW \cup Pre(\overline{v}\overline{f})\,W$, *the behaviour produced by the test process applied to* $\overline{\tau}$ *and the IUT* I *is allowed by the specification* ($t(\overline{\tau}) \in \lfloor M \rfloor$). *Suppose further that there exists some set* T *of states of* M, *that are pairwise distinguished by* W, *such that* $n(T, \overline{v}, \overline{f}) > n' -\mid \hat{T} \mid$. *Then some state of* $Abs(P(M, M_I))$, *reached by* $\overline{v}$ *followed by some non-empty prefix of* $\overline{f}$, *has either been met earlier in* $\overline{f}$ *or is reached by some* $\overline{v}' \in V$.

Proof

A proof by contradiction will be produced: suppose that the following hold.

(1) $n(T, \overline{v}, \overline{f}) > n' -\mid \hat{T} \mid$.
(2) The path through the states of $Abs(P(M, M_I))$, formed by following $\overline{f}$ from the state reached by $\overline{v}$, is cycle free.
(3) No state of $Abs(P(M, M_I))$ met, by following $\overline{f}$ after $\bar{v}$, is reached by a sequence from V.

Let $T = \{s^1, \ldots, s^k\}$ and let $a_j = n(s^j, \overline{v}, \overline{f})$ denote the number of times state s^j is met in $A(M)$ by the path labelled $\overline{f}$ from $s_{\overline{v}}$. Given $s^j \in T$ let R_j denote the set of sequences from $V \cup Pre(\overline{v}\overline{f})\,W$ that reach state s^j.

By Lemma 12, W pairwise distinguish each state reached by sequences in R_j from each state reached by a sequence in R_k for all $s^j, s^k \in T$ with $s^j \neq s^k$. Further, since states of the product machine are not repeated, any two sequences from some R_j, must reach different states of $Abs(M_I)$. Thus, at least $\sum_{s^j \in T} \mid R_j \mid =\mid \hat{T} \mid + \sum_{j=1}^{k} a_j =\mid \hat{T} \mid + n(T, \overline{v}, \overline{f})$ distinct states of $Abs(M_I)$ are met by V and by following $\overline{v}$ by $\overline{f}$. Thus, since $Abs(M_I)$ has at most n' states, $\mid \hat{T} \mid + n(T, \overline{v}, \overline{f}) \leq n'$ and so $n(T, \overline{v}, \overline{f}) \leq n' -\mid \hat{T} \mid$. This provides a contradiction as required. □

Based on this result each state, other than *Fail*, of the product machine is reached by a test in which each $\overline{v}$ contributes a test set of the form of $\cup_{\overline{h} \in \Upsilon_{\overline{v}}} Pre(\overline{v}\overline{h})\, W$ for the set $\Upsilon_{\overline{v}}$ defined in Definition 34.

Definition 34 *Given $\overline{v} \in V$, we require each $\Upsilon_{\overline{v}} \subseteq \Phi^*$ to be a set such that the following properties hold:*

(1) For all $\overline{h} \in \Upsilon_{\overline{v}}$ there exists some set $T \subseteq S$ of states that are pairwise distinguished by W such that $n(T, \overline{v}, \overline{h}) > n' - \mid \hat{T} \mid$.
(2) For all $\overline{f} \in L(A(M))$ there exists $\overline{v} \in V$ and $\overline{h} \in \Upsilon_{\overline{v}}$ such that one of the following holds:
(a) $\overline{v}\overline{h} \in Pre(\overline{f})$
(b) $\overline{f} \in Pre(\overline{v}\overline{h})$

The first property guarantees that no sequences in $\Upsilon_{\overline{v}}$ need be extended further, since extending these sequences is guaranteed to reach states of the product machine already reached. The second property guarantees that the set of sequences in $\Upsilon_{\overline{v}}$ is sufficient: every sequence in $L(A(M))$ is either in some $Pre(\overline{v}\overline{h})$ for some $\overline{h} \in \Upsilon_{\overline{v}}$, and so will be tested, or is an extension of a sequence in some $\{\overline{v}\}\Upsilon_{\overline{v}}$ and so need not be used.

Given $\overline{v} \in V$, $\Upsilon_{\overline{v}}$ is developed by devising a tree in which paths from the root represent walks in $A(M)$ from $s_{\overline{v}}$. A node is a leaf if and only if it satisfies the first condition above: it then need not be extended further. The second condition follows immediately from ϵ being in V and from the way in which the $\Upsilon_{\overline{v}}$ are generated. The algorithm for producing the $\Upsilon_{\overline{v}}$ may be summarised by the following in which Υ denotes the sequences (corresponding to leaves) found so far and Υ^c denotes the sequences current being considered.

(1) Input $\overline{v}$, V, W, and M.
(2) Set $\Upsilon = \varnothing$, $\Upsilon^c = \{< f > |\ (s_{\overline{v}}, f) \in \mathrm{dom}\, F\}$.
(3) Repeat
(4) $\Upsilon = \Upsilon \cup \{\overline{f} \in \Upsilon^c \mid$ there exists a set T of states of M pairwise distinguished by W such that $n(T, \overline{v}, \overline{f}) > n' - \mid \hat{T} \mid\}$.
(5) $\Upsilon^c = \Upsilon^c \setminus \Upsilon$.
(6) $\Upsilon^c = \{\overline{f} g \mid \overline{f} \in \Upsilon^c \wedge (F^*(s_{\overline{v}}, \overline{f}), g) \in \mathrm{dom}\, F\}$.
(7) Until $\Upsilon^c = \varnothing$
(8) Output Υ.

Each iteration of the algorithm involves deciding which elements of Υ^c satisfy the termination criterion, and thus do not need extending, and then extending the others. A sequence $\overline{f}$ to be extended is extended by the relation symbols from Φ that label transitions from the state s of M reached by $\overline{f}$ from $s_{\overline{v}}$.

From now on it will be assumed that the $\Upsilon_{\overline{v}}$ to be used in testing have been chosen, and that each $\Upsilon_{\overline{v}}$ has been produced in the above manner.

The following shows that the $\Upsilon_{\overline{v}}$, as generated above, are finite.

Lemma 19 *For each $\overline{v} \in V$, no sequence in $\Upsilon_{\overline{v}}$ has length greater than $n'n$.*

Proof

Observe that any path of length $n'n$ or more through $A(M)$ must meet some state s at least n' times. The result now follows by observing that the sequence $\overline{f}$ corresponding to this path satisfies the termination criterion with $T = \{s\}$. □

By Lemma 18, if for all $\overline{v} \in V, \overline{h} \in \Upsilon_{\overline{v}}, \overline{f} \in Pre(\overline{v}\overline{h})W$ we have $t(\overline{f}) \in \lfloor M \rfloor$ then the set of $\{\overline{v}\}\Upsilon_{\overline{v}}$ ($\overline{v} \in V$) must reach every reachable state of $Abs(P(M, M_I))$ except, possibly, *Fail*. Thus, if *Fail* is reachable then it is either reached by a sequence in some $\{\overline{v}\}\Upsilon_{\overline{v}}$ or by the extension of one of these sequences by some element of Φ. This leads to testing by applying the test process to each element of the following set:

$$E(V, W, M) = \bigcup_{\overline{v} \in V, \overline{h} \in \Upsilon_{\overline{v}}} Pre(\overline{v}\overline{h})(W \cup \Phi)$$

Test generation may thus proceed via the following algorithm [2].

(1) Choose some deterministic state cover V and characterising set W.
(2) For each $\overline{v} \in V$ generate $\Upsilon_{\overline{v}}$.
(3) Return $E(V, W, M)$.

Theorem 20 *For any choice of deterministic state cover V and characterising set W, the set $E(V, W, M)$ is finite and may be computed.*

Proof

First note that each $\Upsilon_{\overline{v}}$ is found using a search in which a sequence is not extended if it satisfies the condition given in Lemma 18. By Lemma 19 the sequences in the $\Upsilon_{\overline{v}}$ cannot have length greater than nn'. The result thus follows. □

The following results show that if I passes the test then I must be correct.

Theorem 21 *Suppose that the design for test conditions hold and that for all $\overline{f} \in E(V, W, M)$, $t(\overline{f}) \in \lfloor M \rfloor$. Then the state Fail of $Abs(P(M, M_I))$ is not reachable from the initial state of $Abs(P(M, M_I))$.*

[2] The choice of V and W is left to the tester. Note that even when considering a non-deterministic finite state machine, the problem of deciding whether two states can be distinguished is PSPACE-complete [2].

Proof

Proof by contradiction: suppose that for all $\overline{f} \in E(V, W, M)$, $t(\overline{f}) \in \lfloor M \rfloor$ and that the state *Fail* of $Abs(P(M, M_I))$ is reachable. Let $\overline{g} \in \Phi^*$ denote a minimal extension to a sequence in V that reaches the state *Fail* of $Abs(P(M, M_I))$. Note that since $\epsilon \in V$ there is always some such $\overline{g}$: every sequence that reaches *Fail* is an extension of some sequence in V.

Since $\overline{g}$ reaches the state *Fail*, $\overline{g} \in L(Abs(M_I)) \setminus L(A(M))$. By the minimality of $\overline{g}$, $\overline{g} = \overline{g}_1 g$ for some $g \in \Phi$ and $\overline{g}_1 \in L(A(M))$. By the definition of the $\Upsilon_{\overline{v}}$, there are now two possibilities:

(1) There exists $\overline{v} \in V, \overline{h} \in \Upsilon_{\overline{v}}$ such that $\overline{v}\overline{h} \in Pre(\overline{g}_1)$.
(2) There exists $\overline{v} \in V, \overline{h} \in \Upsilon_{\overline{v}}$ such that $\overline{g}_1 \in Pre(\overline{v}\overline{h}) \setminus \{\overline{v}\overline{h}\}$.

In the first case $\overline{g}$ extends $\overline{v}\overline{h}$. By Lemma 18, this contradicts the minimality of $\overline{g}$. In the second case each extension of $\overline{g}_1$ by an element of Φ is in $\Upsilon_{\overline{v}}$. Thus $\overline{g}_1 g$ is contained in the set of sequences from Φ^* to which the test process is applied and so $t(\overline{g}_1 g) \in \lfloor M \rfloor$. By Lemma 16, this contradicts $\overline{g}$ reaching the state *Fail* as required. □

The following shows that if *Fail* is not reachable then I passes the test.

Lemma 22 *Suppose the design for test conditions hold. If the state Fail of $Abs(P(M, M_I))$ is not reachable from the initial state of $Abs(P(M, M_I))$ then for all $\overline{f} \in E(V, W, M)$, $t(\overline{f}) \in \lfloor M \rfloor$.*

Proof

Since M and M_I are completely specified, by Theorem 15, I conforms to M. Thus, all behaviours that may be produced by M_I are allowed by M. The result thus follows. □

Theorem 23 *Suppose the design for test conditions hold. Then the state Fail of $Abs(P(M, M_I))$ is not reachable from the initial state of $Abs(P(M, M_I))$ if and only if for all $\overline{f} \in E(V, W, M)$, $t(\overline{f}) \in \lfloor M \rfloor$.*

Proof

This follows from Theorem 21 and Lemma 22. □

The following is the main result of this paper.

Theorem 24 *If the design for test conditions hold and M and M_I are completely specified then I conforms to M if and only if for all $\overline{f} \in E(V, W, M)$, $t(\overline{f}) \in \lfloor M \rfloor$.*

Proof

This follows from Theorem 23 and Theorem 15. □

Now consider the example. V and W are taken to be the sets $\{\epsilon, <\text{SUB}>\}$ and $\{<\text{SUB}>, <\text{SUBR}>\}$ respectively and $S_V = \{S0, S1\}$. There is no sequence that distinguishes $S1$ and $S2$ since they are not pairwise distinguishable. Therefore the maximal pairwise distinguishable sets of states are $A = \{S0, S1, S3\}$ and $B = \{S0, S2, S3\}$. These will be used in constructing the sequences of F_i.

It will be assumed that the IUT contains at most four states, so $n' = 4$. For each state in S_V, the sequences in $\Upsilon_{\overline{v}}$ must be constructed. The process can be considered as determining for each state $s \in S_V$ a set of paths, in a tree rooted on s, for which the leaves are determined by state counting. In this case there are two states to consider: $S0$ and $S1$. In generating the $\Upsilon_{\overline{v}}$, a node n_i of the tree formed is a leaf if and only if at least one of the following hold:

(1) $n' - \mid \hat{A} \mid +1 = 3$ or more nodes on the path from the root to n_i represent states that are contained in A.
(2) $n' - \mid \hat{B} \mid +1 = 4$ or more nodes on the path from the root to n_i represent states contained in B.

In each case the root is not included in this calculation.

To provide an illustration of the process involved, the expansion of the tree for Υ_ϵ is shown in Figure 4. For each leaf n_i, this shows the set, or sets, through which it may be shown that n_i is a leaf.

In the tree, each node has a corresponding unique state of the specification. This is guaranteed since the associated automaton $A(M)$ is deterministic. Where a node with state s as label is not a leaf, it is extended by each relation from Φ that labels a transition from state s in M. These labels are not shown in Figure 4 in order to simplify the figure. Where multiple transitions go between two states, this is represented by a single edge as explained in Figure 4. Since there is a transition with relation *Sub* from $S0$ to $S1$, for each node n that is not a leaf and has label $S0$, there is an edge that has label *Sub* from n to some node n' with label $S1$. An example of this is the edge from the root to a node with label $S1$.

A node n is a leaf if the path from the root to n satisfies the termination criterion that defines the $\Upsilon_{\overline{v}}$; 3 or more nodes (after the root) on the path from the root to n_i represent states that are contained in A or 4 or more nodes on the path from the root to n_i represent states contained in B. In Figure 4 an arrow from letter A to a leaf indicates that the node is a leaf for the first of these reasons while an arrow from letter B to a leaf indicates

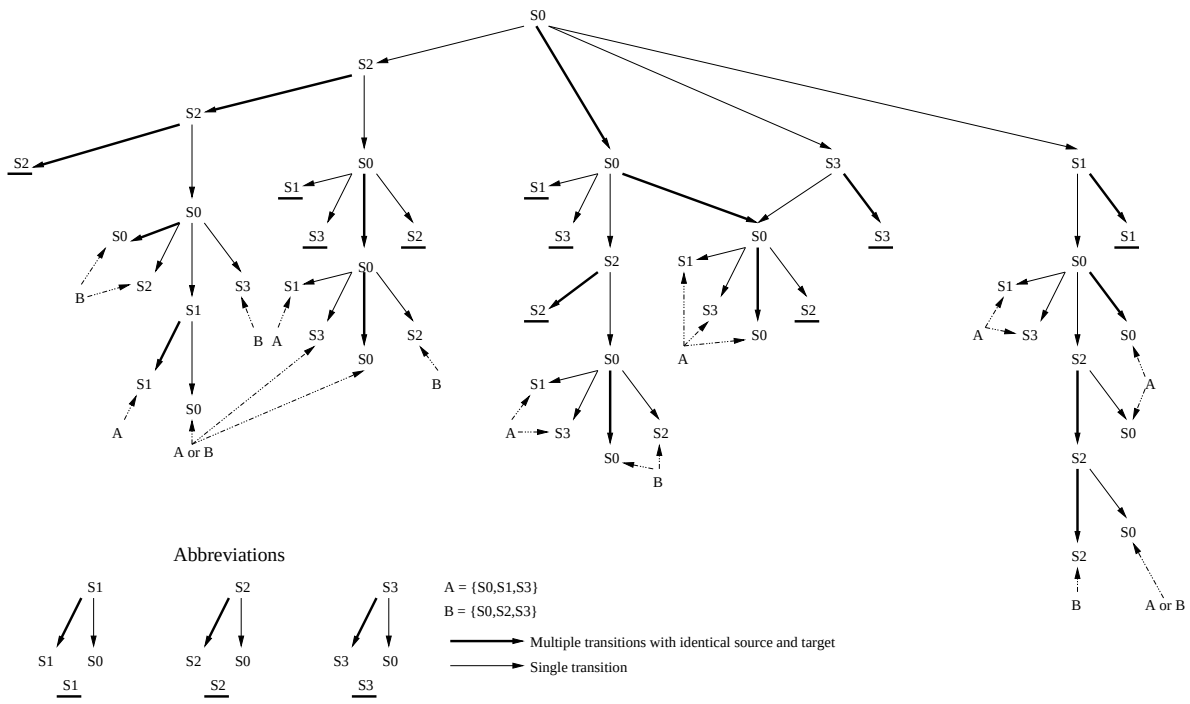

Fig. 4. The Tree Expansion for Υ_ϵ

that the node is a leaf for the second reason. Observe that some nodes may be leaves under both criteria. An example of this is the leaf that is reached by the path $S0 \rightarrow S2 \rightarrow S2 \rightarrow S0 \rightarrow S1 \rightarrow S0$.

Each leaf represents the sequence from Φ^* generated by following the path from the root to that leaf. For example, the path $S0 \rightarrow S1 \rightarrow S0 \rightarrow S1$ represents the sequence $< Sub, Print, Sub >$. Υ_ϵ is the set of such sequences.

Note that if the brute-force approach is applied, without using V or W, we get the test set $\mathcal{T}_{n'n} = \Phi^{n'n}$ which contains 13^{16} sequences (since we test from the completely-specified stream X-machine given in Figure 2). Clearly, far fewer test sequences are produced using state counting.

9 Observations and Future Work

It is worth noting that if all states of M are d-reachable and pairwise distinguishable then this method reduces to the W-method [7]. Generally, increasing the size of either V or W reduces the test number of test sequences required. Naturally the condition, that all states of M are d-reachable and pairwise distinguishable might be seen as a further design for test condition. Interestingly, this condition does not require the processing relations of M to be deterministic. Instead, it requires that there are sufficient implemented sequences from

Φ^* to reach and pairwise distinguish the states of M. These sequences themselves may contain non-determinism: they may contain relations. Thus, it is possible for all of the states of M to be d-reachable and pairwise distinguishable without there being any (non-empty) input sequence for which there is only one allowed output sequence.

Since the implementation is deterministic, its response to an input sequence $\overline{x}$ is fixed: if $\overline{x}$ is input again, in the initial state, the same output sequence will be produced. This property has been utilised, in order to reduce the test effort, when testing a deterministic implementation against a non-deterministic finite state machine [15]. Potentially, it might also be used to reduce the effort when testing a deterministic implementation against a non-deterministic stream X-machine.

Suppose some state $s \in S$ of M is not d-reachable. Suppose further that, in testing, some input sequence $\overline{x}$ triggers an output sequence $\overline{y}$ with the property that, in M, $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$ and $F^*(s_0, \overline{f}) = s$. Then s may now be considered to be d-reachable by $\overline{f}$ and so s may be added to S_V and $\overline{f}$ may be added to V. This suggests that the test might be adaptive: the tests applied depend upon the results of earlier tests.

The behaviour observed during testing may also help the tester to reason about the states of $P(M, M_I)$ that are met. For example, while two states s'_1 and s'_2 of M_I may conform to the same state s of M, they may be distinguished during testing. The problem of finding ways of exploiting such information, in order to reduce the test effort, will be a topic of future work.

This paper has assumed that M and M_I are completely specified. While this is often a reasonable assumption, it is worth considering how this condition may be relaxed. It seems likely that a test generation algorithm might be similar to that outlined in this paper. However, by Theorem 7, an additional element would be required. This would check that $\mathrm{dom}\, M = \mathrm{dom}\, M_I$. This could be achieved by checking that the union of the input domains of the relations executable from a state of the product machine is the same as that for the corresponding state of M. Tests might be used to check this property at each state of the product machine reached in the search.

Finally, it should be possible to weaken the assumption that Φ is observable with respect to $\mathcal{T}E$. Instead, it should be sufficient to assume that for each $f \in \Phi, m \in \mathcal{M}$, $x \in U_f$, and $y \in Out$ such that there exists $m' \in Mem$ with $((m, x), (m', y)) \in f$, there are only a finite number of possible memory values after output y has been observed in response to the input of x when the memory is m. This assumption might lead to a test that considers all possible memory values after an observed behaviour.

10 Conclusions

This paper has introduced a test generation algorithm for testing a deterministic implementation against a non-deterministic stream X-machine. The test produced is guaranteed to determine correctness under certain design for test conditions.

The situation addressed is one in which the implementation is deterministic and the specification is non-deterministic. This is important because, while implementations are typically deterministic, non-determinism aids abstraction and so is often deployed in specifications. This case has not been previously covered in the literature.

Previous work on testing against a stream X-machine has utilised the W-method. This is possible because, in the cases previously considered, for the implementation to be correct it must have the same structure as the specification. Here, however, the implementation may conform to the specification and yet have a significantly different structure. Instead of using the W-method, test generation has been based on the product machine: testing can be represented as a process of executing the implementation under test in order to decide whether the special state *Fail* of the product machine is reachable.

Where the implementation is deterministic, test output may provide information that can be used to further reduce the test effort. Such adaptive testing will form a topic for future research.

Acknowledgments

We would like to thank the anonymous reviewers, whose many useful comments significantly strengthened this paper. This work was supported by EPSRC grants: GR/R43150, Formal Methods and Testing (FORTEST) and GR/R98938/01, Testability Transformation (TeTra).

References

[1] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. In *Protocol Specification, Testing, and Verification VIII*, pages 75–86, Atlantic City, 1988. Elsevier (North-Holland).

[2] R. Alur, C. Courcoubetis, and M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *27th ACM Symposium on Theory of Computing*, pages 363–372, 1995.

[3] T. Balanescu, H. Georgescu, M. Gheorghe, and C. Vertan. Communicating stream X-machines systems are no more than X-machines. In *Twelfth International Symposium on Fundamentals of Computation Theory (FCT'99)*, Iasi, Romania, September 1999.

[4] J. Barnard. COMX: A design methodology using communicating X-machines. *Information and Software Technology*, 40(5-6):271–280, 1998.

[5] J. Barnard, J. Whitworth, and M. Woodward. Communicating X-machines. *Information and Software Technology*, 38(6):401–407, 1996.

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

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

[8] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, London, 3 edition, 1972.

[9] J. Derrick and E. Boiten. Testing refinements of state-based formal specifications. *Journal of Software Testing, Verification, and Reliability*, 9(1):27–50, 1999.

[10] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93, First International Symposium on Formal Methods in Europe*, pages 268–284, Odense, Denmark, 19-23 April 1993. Springer-Verlag, Lecture Notes in Computer Science 670.

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

[12] M. C. Gaudel. Testing can be formal too. *Lecture Notes in Computer Science*, 915:82–96, 1995.

[13] D. Harel and M. Politi. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, New York, 1998.

[14] R. M. Hierons. Testing from a Z specification. *Journal of Software Testing, Verification and Reliability*, 7(1):19–33, 1997.

[15] R. M. Hierons. Adaptive testing of a deterministic implementation against a nondetermistic finite state machine. *The Computer Journal*, 41(5):349–355, 1998.

[16] R. M. Hierons and M. Harman. Testing comformance to a quasi-non-determinstic stream X-machine. *Formal Aspects of Computing*, 12(6):423–442, 2000.

[17] R. M. Hierons and H. Ural. Reduced length checking sequences. *IEEE Transactions on Computers*, 51(9):1111–1117, 2002.

[18] R.M. Hierons, T.-H. Kim, and H. Ural. Expanding an extended finite state machine to aid testability. In *IEEE Annual Computer Software and Applications Conference (COMPSAC 2002)*, pages 334–339, Oxford, England, August 2002.

[19] M. Holcombe. X-machines as a basis for dynamic system specification. *Software Engineering Journal*, 3(2):69–76, 1988.

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

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

[22] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63(3&4):159–178, 1997.

[23] F. Ipate and M. Holcombe. A method for refining and testing generalised machine specifications. *Journal of Software Testing, Verification and Reliability*, 8(2):61–81, 1998.

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

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

[26] ITU-T. *Recommendation Z.100 Specification and description language (SDL)*. International Telecommunications Union, Geneva, Switzerland, 1999.

[27] E. Kehris, G. Eleftherakis, and P. Kefalas. Using X-machines to model and test discrete event simulation programs. In N. Mastorakis, editor, *Systems and Control: Theory and Applications*, pages 163–171. World Scientific and Engineering Society Press, Athens, 2000.

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

[29] E. P. Moore. Gedanken-Experiments. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.

[30] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das. Nondeterministic state machines in protocol conformance testing. In *Proceedings of Protocol Test Systems, VI (C-19)*, pages 363–378, Pau, France, 28-30 September 1994. Elsevier Science (North-Holland).

[31] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In *Testing of Communicating Systems, IFIP TC6 9th International Workshop on Testing of*

Communicating Systems, pages 125–141, Darmstadt, Germany, 9-11 September 1996. Chapman and Hall, London.

[32] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[33] A. W. Tomita and K. Sakamura. Improving design dependability by exploiting an open model-based specification. *IEEE Transactions on Computers*, 48(1):24–37, 1999.

[34] H. Ural, K. Saleh, and A. Williams. Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications*, 23:609–627, 2000.

[35] H. Ural, X. Wu, and F. Zhang. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99, 1997.

[36] M. U. Uyar and A. Y. Duale. Resolving inconsistencies in EFSM modeled specifications. In *IEEE Military Communications Conf. (MILCOM)*, Atlantic City, NJ, October 1999.

[37] N. V. Yevtushenko, A. V. Lebedev, and A. F. Petrenko. On checking experiments with nondeterministic automata. *Automatic Control and Computer Sciences*, 6:81–85, 1991.