Under consideration for publication in Formal Aspects of Computing

Testing conformance to a quasi-nondeterministic stream X-machine

R.M. Hierons and M. Harman
Brunel University

Abstract. Stream X-machines have been used in order to specify a range of systems. One of the strengths of this approach is that, under certain well defined conditions, it is possible to produce a finite test that is guaranteed to determine the correctness of the implementation under test (IUT). Initially only deterministic stream X-machines were considered in the literature. This is largely because the standard test algorithm relies on the stream X-machine being deterministic.

More recently the problem of testing to determine whether the IUT is *equivalent* to a nondeterministic stream X-machine specification has been tackled ([17]). Since nondeterminism can be important for specifications, this is an extremely useful extension. In many cases, however, we wish to test for a weaker notion of correctness called *conformance*. This paper considers a particular form of nondeterminism, within stream X-machines, that shall be called quasi-nondeterminism. It then investigates the generation of tests that are guaranteed to determine whether the IUT *conforms* to a quasi-nondeterministic stream X-machine specification. The test generation algorithm given is a generalization of that used for testing from a deterministic stream X-machine.

keywords: Stream X-machine, nondeterminism, quasi-nondeterminism, testing, design for test conditions, conformance.

1. Introduction

Formal specifications and models can help in the production of high quality software. They eliminate the opportunity for ambiguity and allow the application of, possibly automated, formal analysis. Even where a formal specification or

Correspondence and offprint requests to: R.M. Hierons and M. Harman
Brunel University

model is used it is, however, important to test the implementation ([6]). Where there is a formal specification or model, this may be used as the basis of test automation (see, for example, [5, 8, 11, 10, 12]).

One approach, to formally specifying a system, is to use a form of extended finite state machine called a stream X-machine ([13, 14, 3, 18, 2, 15, 16, 1]). A stream X-machine describes a system as a finite set of logical states, each with an internal store or memory, with transitions between the logical states. A transition is triggered by an input value, produces an output value and may alter the internal store and logical state. A stream X-machine may be modelled by a finite automaton in which the arcs are labelled by function names. Stream X-machines will be described in Section 3.

Where complex systems are being developed testing is often expensive and sometimes ineffective. While the presence of a formal model or specification, of the required behaviour, may allow test generation to be automated it is still often difficult to deduce much from the implementation under test (IUT) behaving correctly on the test set produced. Thus we have the following problems.

1. How can we produce test sets that are capable of providing a high degree of confidence in the correctness of the IUT?
2. How can we simplify the problem of test generation to allow automation?

It is possible to approach both problems by developing testable specifications and designs. Thus, testing is considered throughout the development lifecycle, not just at the end. The reduction in the cost of testing and the increase in test effectiveness often justifies any extra expense created by introducing testability. This approach, in which testability is designed into a system, is particularly prevalent in the design and test of hardware components (see, for example, [21]) but has been proposed in other areas (see, for example, [23]).

The standard (automated) test generation algorithm used with stream X-machines assumes that certain conditions, called *design for test conditions*, hold. Where the design for test conditions hold the test generated by this method is *guaranteed* to determine the correctness of the IUT ([18]). This integration of specification and test generation is one of the most significant benefits of using deterministic stream X-machines in software development.

The use of nondeterminism can aid abstraction and thus is highly appropriate for specifications. In Section 4 quasi-nondeterministic stream X-machines will be described. Essentially these are stream X-machines that allow nondeterminism to be introduced in certain ways. Recently the test generation algorithm, for testing against a deterministic stream X-machine, has been adapted to testing against a nondeterministic stream X-machine ([17]). This paper gives an algorithm for generating a test that determines whether the IUT is *equivalent* to a nondeterministic stream X-machine specification.

In many situations the notion of correctness used in [17] is too strong. Instead it is sufficient for the IUT to be defined on all input upon which the specification is defined and for the behaviour contained in the IUT to be a subset of the behaviour contained in the specification. This notion of correctness is often called *conformance.* Naturally, equivalence and conformance coincide when the specification is deterministic. One important special case in which conformance, rather than equivalence, is required is where the IUT is deterministic but the specification is nondeterministic. This special case is quite common as, while specifications are often nondeterministic, actual implementations are usually deterministic

This paper considers the case in which an IUT I is tested, for conformance, against a quasi-nondeterministic stream X-machine M that specifies the required behaviour. In Section 6 the design for test conditions, expected of deterministic stream X-machines, are generalized to form design for test conditions for a quasi-nondeterministic stream X-machine. Section 7 then gives a test generation algorithm that produces a test that determines the correctness of the IUT I as long as M and I satisfy the design for test conditions. The test generation algorithm is more general than, and as powerful as, those given for testing against a deterministic stream X-machine. The test generation algorithm and notion of correctness used is also different from that used in [17]. A proof of correctness of the test algorithm is given in Section 8. This is followed by a discussion of future work and finally conclusions are drawn.

2. Finite Automata and Finite State Machines

A finite automaton (FA) M is defined by a tuple (S, s_1, h, Σ, T) in which S is the finite set of *states*, s_1 is the *initial state*, h is the *transition function*, Σ is the finite *input alphabet* and T is the set of *final states*. The function h gives the set of states that M may move to given a current state and input. Thus if M is in state s and receives input x it moves to some state from the set $h(s, x) \subseteq S$. The function h may be extended, to take input sequences, to give h^*. If ϵ denotes the empty sequence then h^* is defined by:

$$\forall\, s \in S.h^*(s, \epsilon) = \{s\}$$

$$\forall\, s \in S, \overline{x} \in \Sigma^*, x' \in \Sigma.h^*(s, \overline{x}x') = \bigcup_{s' \in h^*(s,\overline{x})} h(s', x')$$

Throughout the paper any variable name with a bar over it represents a sequence.

FA M *accepts* $\overline{x} \in \Sigma^*$ if and only if $\overline{x}$ can take M from the initial state to some final state: $h^*(s_1, \overline{x}) \cap T \neq \varnothing$. The set of strings accepted by M defines a *language* $L(M) = \{\overline{x} \in \Sigma^* \mid h^*(s_1, \overline{x}) \cap T \neq \varnothing\}$. Similarly, the set of sequences that can reach a final state from state s of M form the language $L_M(s) = \{\overline{x} \in \Sigma^* \mid h^*(s, \overline{x}) \cap T \neq \varnothing\}$. Thus, $L(M) = L_M(s_1)$.

Consider, for example, the FA M_1 given in Figure 1 in which the final state s_3 is denoted by a double circle. Here, $S = \{s_1, s_2, s_3\}$, $\Sigma = \{a, b\}$ and $T = \{s_3\}$. If M_1 receives input a while in state s_1 it may either move to state s_2 or stay in state s_1. Thus, $h(s_1, a) = \{s_1, s_2\}$. M_1 defines the language $L(M_1)$ of strings composed of elements of Σ that end in ab.

FA M is said to be *initially connected* if, for every state $s_i \in S$ there is some input sequence $\overline{x}$ that can reach s_i: $s_i \in h^*(s_1, \overline{x})$. M is said to be *strongly connected* if for every ordered pair of states (s_i, s_j), $s_i, s_j \in S$, there is some input sequence $\overline{x}$ that can move M from s_i to s_j: $s_j \in h^*(s_i, \overline{x})$. It is easy to check that M_1 is initially connected but not strongly connected.

Input sequence $\overline{x}$ *distinguishes* states s_i and s_j of M if and only if $\overline{x}$ is contained in one and only one of $L_M(s_i)$ and $L_M(s_j)$. Given sets A and B let $A \bigtriangleup B = (A \setminus B) \cup (B \setminus A)$. Thus $\overline{x}$ distinguishes s_i and s_j if and only if $\overline{x} \in L_M(s_i) \bigtriangleup L_M(s_j)$. In M_1, for example, b distinguishes s_1 and s_2. If there is

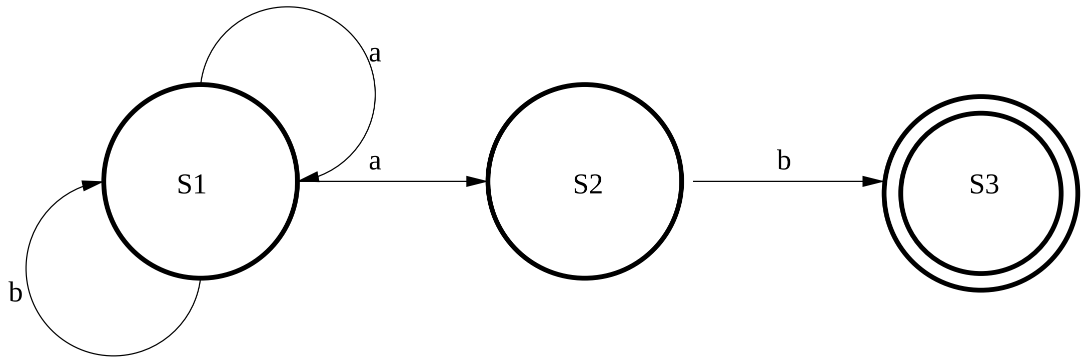

Fig. 1. The Finite Automaton M_1

some $\overline{x}$ that distinguishes s_i and s_j then s_i and s_j are said to be *distinguishable.* If s_i and s_j are not distinguishable they are said to be *equivalent.* Two FA are *equivalent* if their initial states are equivalent.

FA M is *deterministic* if for every state s and input x there is at most one transition that leaves s and is labelled with x: $\forall s \in S, x \in \Sigma.(s,x) \in dom\ h \Rightarrow \mid h(s,x) \mid = 1$. FA M is said to be *nondeterministic* if it is not deterministic. A deterministic FA M is said to be *minimal* if there is no equivalent deterministic FA with fewer states than M. A deterministic FA M is minimal if it is initially connected and no two states of M are equivalent.

Given a nondeterministic FA M, it is possible to derive an equivalent deterministic FA M' ([22]). Given a deterministic FA M it is possible to produce some equivalent minimal deterministic FA M' ([20]). Thus, given any FA M it is possible to produce a minimal deterministic FA M' that is equivalent to M. The nondeterministic FA M_1 given in Figure 1 is, for example, equivalent to the minimal deterministic FA M_2 given in Figure 2. In contrast, it will transpire that given a quasi-nondeterministic stream X-machine, there may be no equivalent deterministic stream X-machine. This is because a quasi-nondeterministic stream X-machine may allow more than one output sequence in response to an input sequence and no deterministic stream X-machine may describe such behaviour.

Suppose M is a deterministic FA. A set W of input sequences is said to be a *characterizing set* for M if for every pair (s_i,s_j) of states of M, there is some $\overline{w} \in W$ that distinguishes s_i and s_j. Every minimal deterministic FA has a characterizing set ([9]). M_2, for example, has characterizing set $\{\epsilon, b\}$.

A *finite state machine (FSM) M* is a FA in which every state is a final state and each transition has an associated output value. An FSM may be represented by an associated FA in which the arcs are labelled by input/output pairs.

3. Stream X-machines

X-machines are essentially FA in which the arcs represent relations and there is a basic set on which the relations are defined. The allowed behaviours are given by the sequences of arcs defined by walks from initial states to final states. A stream X-machine M is an X-machine in which the basic set X is composed of

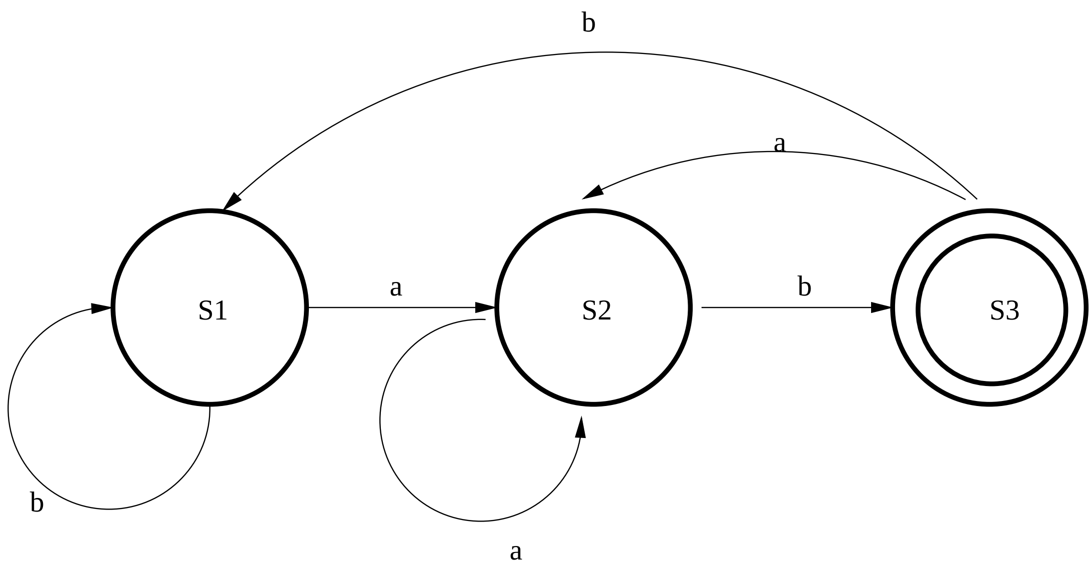

Fig. 2. The Finite Automaton M_2

an internal memory, an input stream and an output stream and the functions of M act on X. Suppose *Mem* denotes the set of memory values, *In* denotes the set of input values and *Out* denotes the set of output values. Given set A let A^* denote the set of sequences of elements of A. Then X is $In^* \times Mem \times Out^*$.

At the beginning of computation the output stream is empty and the input stream contains the full input of the execution instance being considered. If M reaches the situation in which it is in a final state and the input stream is empty then the computation has succeeded and the output stream contains the values output by M. Otherwise the computation has not completed and the input sequence is not part of the input domain of M. In this case the system halts and the string $\overline{y}$, contained in the output stream the last time a final state was reached, is followed by failure ($\perp$), producing output $\overline{y} \perp$. If no final state has been reached, the output is $\perp$. The failure $\perp$ represents halting and possibly the production of an error message and is not explicitly contained in the output domain of M.

Definition 1. A stream X-machine is defined by a tuple $(In, Out, S, Mem, \Phi, F, s_1, m_0, T)$ in which ([15])

1. In is the finite *input alphabet.*
2. Out is the *output alphabet.*
3. S is the finite set of *states.*
4. Mem is the *memory.* Mem need not be finite.
5. Φ is the set of *processing functions.* Each function has type $Mem \times In \rightarrow Out \times Mem$.
6. F is the partial *next state function* with type $S \times \Phi \rightarrow \mathcal{P}(S)$.
7. s_1 is the *initial state.*

8. m_0 is the *initial memory value.*
9. T is the set of *final states.*

When considering the problem of testing from a stream X-machine it is normal to assume that all states are final states: $T = S$ ([15]). This is not a major limitation when modelling an interactive system, which responds to each input value separately.

Given a sequence $\overline{f}$ of functions from Φ, an associated function $\|\overline{f}\|$ may be produced by combining the functions in $\overline{f}$. Then $\|\overline{f}\|$ takes a memory value and an input string of length $|\ \overline{f}\ |$ and returns a memory value and an output sequence of length $|\ \overline{f}\ |$. If $f \in \Phi$ and $\overline{f} \in \Phi^*$ then

Definition 2.

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

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

It is worth noting that this definition of $\|\overline{f}\|$ holds for relations, as well as functions.

The sequence $\overline{f}$ defines the input/output relation $\langle\overline{f}\rangle$ defined by

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

Abusing this notation, given a walk $\overline{w} = a_1, \ldots, a_r$ from M, a_i having associated function f_i, $\langle\overline{w}\rangle$ shall denote $\langle f_1, \ldots, f_r\rangle$.

Let $W(s_i, s_j)$ denote the set of walks from s_i to s_j and $W^T = \bigcup_{s_j \in T} W(s_1, s_j)$. The behaviour of M is defined by the functions given by the set of walks from initial states to final states. M describes the relation defined by the following.

Definition 3.

$$\lfloor M \rfloor = \bigcup_{\overline{w} \in W^T} \langle\overline{w}\rangle\,.$$

Then it is possible to say that M has an input domain; that of $\lfloor M \rfloor$. Thus $dom\ M = dom\ \lfloor M \rfloor$. Given stream X-machine M and input sequence $\overline{x} \in In^*$, M relates $\overline{x}$ to the output sequences in

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

Given $\overline{x} \notin dom\ M$, $\overline{x} = \overline{x}_1\overline{x}_2$ for some maximal $\overline{x}_1 \in dom\ M$, let $\overline{x}_1$ be denoted $pre_M(\overline{x})$. Where no substring of $\overline{x}$ is in $dom\ M$, $pre_M(\overline{x}) = \epsilon$. It is possible to complete $\lfloor M \rfloor$ by adding the pair $(\overline{x}, \overline{y}\perp)$ for each $\overline{x} \in In^* \setminus dom\ M$, $(pre_M(\overline{x}), \overline{y}) \in \lfloor M \rfloor$. This gives the relation $\lfloor M \rfloor_\perp$, of type $In^* \leftrightarrow (Out \cup \{\perp\})^*$, defined by the following.

Definition 4.

$$\lfloor M \rfloor_\perp = \lfloor M \rfloor \cup \{(\overline{x}, \overline{y}\perp) \mid \overline{x} \in In^* \setminus dom\ M \wedge (pre_M(\overline{x}), \overline{y}) \in \lfloor M \rfloor\}$$

It is worth noting that the semantics give $\lfloor M \rfloor$ to be defined in terms of the set of walks from the initial state to final states. Thus, any rewrite of M that preserves the set of sequences of functions on walks from the initial state to final states preserves the meaning of M. Thus, for example, the X-machines shown in Figure 3 are equivalent. Those interested in label transition systems will note

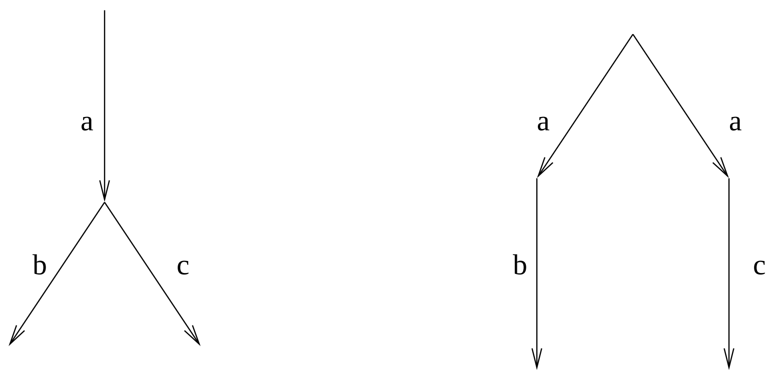

Fig. 3. Two Label Transition System

that if these denoted label transition systems they would not, in general, be equivalent. This is because:

1. in the first having performed a the system moves to a state in which it is capable of performing either b or c;
2. in the second having performed a the system must either be incapable of performing b or be incapable of performing c.

The form of equivalence used with finite automata, finite state machines and X-machines is often called *trace equivalence*.

4. Quasi-nondeterministic stream X-machines

Since nondeterminism can aid abstraction, it is highly appropriate for specifications. Therefore, allowing the use of nondeterminism should extend the applicability of the stream X-machine development method. In this paper two sources of nondeterminism shall be allowed. These two forms of nondeterminism shall now be described. This shall be followed by a definition of a *quasi-nondeterministic stream X-machine*.

The definition of a stream X-machine allows more than one next state for some state s and function f with $(s, f) \in dom\ F$. Here, the execution of f in state s may lead to any state from $F(s, f)$. This form of nondeterminism, in which there is more than one possible next state for some (s, f), shall be called state nondeterminism.

Definition 5. M has *state nondeterminism* if there exist $s \in S$, $f \in \Phi$ with $(s, f) \in dom\ F$ and $\mid F(s, f) \mid > 1$.

As noted earlier, any rewrite of an X-machine or stream X-machine M that preserves $L(M)$ preserves the meaning of M. Thus, this form of nondeterminism may be removed by rewriting the stream X-machine using standard algorithms that take a nondeterministic finite automaton and produce an equivalent deterministic finite automaton. Allowing F to return a set of next states may, however, be used to produce a more compact and understandable model. Consider, for example, the two equivalent finite automata given earlier in Figures 1 and 2.

It is possible to generalize the definition of a stream X-machine by allowing operators from Φ to be relations rather than functions. This form of nondeterminism shall be called operator nondeterminism.

Definition 6. M has *operator nondeterminism* if some element of Φ is a relation but not a function.

There seems no good reason to outlaw operator nondeterminism and, as shall be seen later, it can be useful. Thus, operator nondeterminism shall be allowed for quasi-nondeterministic stream X-machines. Unlike state nondeterminism, operator nondeterminism cannot be removed by rewriting the stream X-machine.

Allowing state nondeterminism and operator nondeterminism gives the following definition of a quasi-nondeterministic stream X-machine.

Definition 7. A *quasi-nondeterministic stream X-machine* is defined by a tuple $(In, Out, S, Mem, \Phi, F, s_1, m_0, T)$ in which

1. In is the finite *input alphabet.*
2. Out is the *output alphabet.*
3. S is the finite set of *states.*
4. Mem is the *memory.* Mem need not be finite.
5. Φ is the set of *processing relations.* Each relation has type $Mem \times In \leftrightarrow Out \times Mem$.
6. F is the partial *next state function* with type $S \times \Phi \rightarrow \mathcal{P}(S)$.
7. s_1 is the *initial state.*
8. m_0 is the *initial memory value.*
9. T is the set of *final states.*

and for all $s \in S$ and $f, f' \in \Phi$ such that $(s, f), (s, f') \in dom\ F, f \neq f' \Rightarrow dom\ f \cap dom\ f' = \{\}$.

The last condition says that any two different operators that may be applied in some state s must have disjoint domains. Where necessary, this final condition may be guaranteed by adding a special (unique) value to the input domain for each operator. These special inputs may be kept (but hidden from the user) or removed before the system is deployed. It is worth noting that the definitions of a nondeterministic stream X-machine given in ([17]) allow operator but not state nondeterminism. The definitions are otherwise equivalent.

Throughout this paper it will be assumed that the specification is a quasi-nondeterministic stream X-machine M and that the implementation I behaves like some unknown quasi-nondeterministic stream X-machine M_I.

Consider the Vending Stream X-Machine pictured in Figure 4. The vending machine has BUTTONS, SLOTS and Lights. Buttons are input devices used to select a particular behaviour. The $\mathcal{M}$ button, is a manager button, which is key operated. The $\mathcal{V}$ button requests the vending of a chocolate. This happens if sufficient payment has been received (the chocolate costs 20). The $\mathcal{C}$ button requests change to be returned to the user (subject to the amount of credit in the machine). The $\mathcal{E}$ button is used to empty the machine of change. The $\mathcal{U}$ button is used to leave management mode and return to user mode.

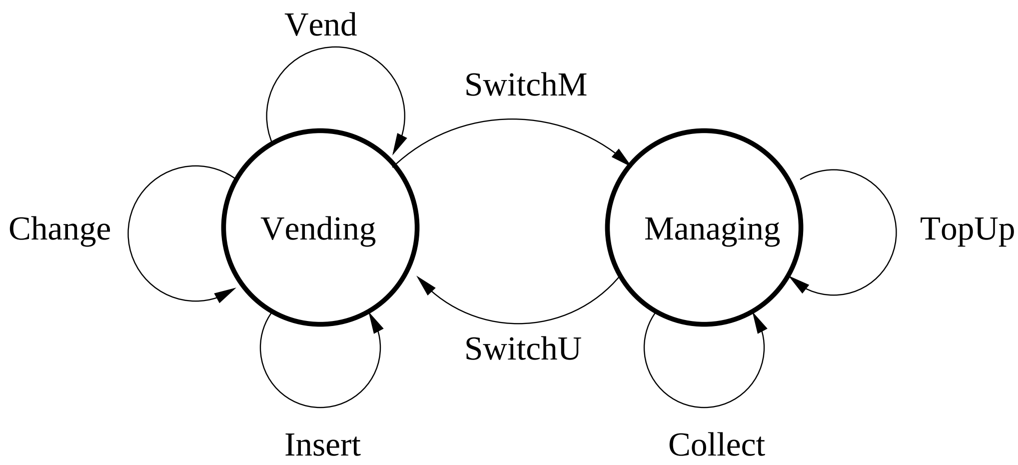

Fig. 4. The Vending Stream X-Machine

Four lights, Choc, NoVend, $\mathsf{ManageOn}$, $\mathsf{ManageOff}$ indicate machine responses. Choc indicates that a chocolate has been dispensed. NoVend indicates that a chocolate was requested but cannot be delivered. $\mathsf{ManageOn}$, indicates that the machine is switching over to management mode and $\mathsf{ManageOff}$ indicates that the machine is switching from management mode to user mode. These lights illuminate for a period of two seconds.

The machine has four slots, $\textsc{UserIn}$, $\textsc{UserOut}$, $\textsc{ManagerIn}$, $\textsc{ManagerOut}$, through which money is inserted and returned to the user and manager respectively.

The machine also has an LCD display which illuminates for up to four seconds, displaying the amount of credit the machine possesses when the user inserts coins.

The slots into which the manager inserts change and from which change is returned to the manager are not normally exposed. When the manage button, $\mathcal{M}$, is pressed (using the key), a panel opens at the back of the machine, making these two additional slots available. In addition there is an LCD display behind the panel which indicates the value of the money stored in the machine (for up to four seconds) each time a coin is inserted into the manager slot.

This is formalised in Z below:

$Coins ::= 10 \mid 20 \mid 50$

$BUTTONS ::= \mathcal{V} \mid \mathcal{C} \mid \mathcal{M} \mid \mathcal{E} \mid \mathcal{U}$

$\mathsf{Lights} ::= \mathsf{Choc} \mid \mathsf{NoVend} \mid \mathsf{ManageOn} \mid \mathsf{ManageOff}$

$\textsc{Slots} ::= \textsc{UserIn} \mid \textsc{UserOut} \mid \textsc{ManagerIn} \mid \textsc{ManagerOut}$

$$
\begin{array}{|l}
\textit{State} \\
\hline
\textit{Bank} : \textit{Coins} \rightarrow \mathbb{N} \\
\textit{Credit} : \mathbb{N} \\
\hline
\\
\hline
\end{array}
$$

The initial state is defined by the Schema below:

$$
\begin{array}{|l}
\textit{InitialState} \\
\hline
\Delta \textit{State} \\
\hline
\textit{Credit}' = 0 \\
\textit{Bank}' = \{(x, 0) \mid x \in \textit{Coins}\} \\
\hline
\end{array}
$$

A function **Value** is defined, which determines the value of a coin-bag:

$$
\begin{array}{|l}
\textit{Value} : (\textit{Coins} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\
\hline
\textit{Value}(b) = 10 * b(10) + 20 * b(20) + 50 * b(50)
\end{array}
$$

Two other operations on bags of coins are required, which extend the associated set operators to coin bag operators:

$$
\begin{array}{|l}
\_ \subseteq \_ : (\textit{Coins} \rightarrow \mathbb{N}) \times (\textit{Coins} \rightarrow \mathbb{N}) \rightarrow \textit{bool} \\
\hline
a \subseteq b \Leftrightarrow \forall\, c \in \textit{Coins}.a(c) \leq b(c)
\end{array}
$$

$$
\begin{array}{|l}
\_ - \_ : (\textit{Coins} \rightarrow \mathbb{N}) \times (\textit{Coins} \rightarrow \mathbb{N}) \rightarrow (\textit{Coins} \rightarrow \mathbb{N}) \\
\hline
a \subseteq b \Rightarrow a - b = \{c \mapsto a(c) - b(c) \mid c \in \textit{Coins}\}
\end{array}
$$

The User functions **Insert**, **Vend** and **Change** are defined as follows:

$$
\begin{array}{|l}
\textit{Insert} \\
\hline
i? : (\textit{Coins} \times \textit{Slots}) \\
\Delta \textit{State} \\
r! : \mathbb{N} \\
\hline
i? = (c, \textrm{U}\small{\textrm{SER}}\textrm{I}\small{\textrm{N}}) \\
\textit{Credit}' = \textit{Credit} + c \\
\textit{Bank}' = \textit{Bank} \oplus \{c \mapsto \textit{Bank}(c) + 1\} \\
r! = \textit{Credit}' \\
\hline
\end{array}
$$

The function *Vend* allows the customer to obtain a chocolate if sufficient money has been put in the machine (the credit is sufficient) and the machine is capable of providing change after this purchase.

$$\begin{array}{|l}
\hline
\textit{Vend} \\
i? : BUTTONS \\
\Delta State \\
r! : \mathsf{Lights} \\
\hline
i? = \mathcal{V} \\
(\exists\, b \subseteq Bank \bullet Value(b) = Credit - 20) \\
\Rightarrow \\
Credit' = Credit - 20 \land r! = choc \land Bank' = Bank \\
\\
\neg(\exists\, b \subseteq Bank \bullet Value(b) = Credit - 20) \\
\Rightarrow \\
Credit' = Credit \land Bank' = Bank \land r! = novend \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Change} \\
i? : BUTTONS \\
\Delta State \\
chg! : Coins \rightarrow \mathbb{N} \\
r! : ((Coins \rightarrow \mathbb{N}) \times \textsc{Slots}) \\
\hline
i? = \mathcal{C} \\
Credit' = 0 \\
Bank' = Bank - chg! \\
chg! \subseteq Bank \\
Value(chg!) = Credit \\
r! = (chg!, \textsc{UserOut}) \\
\hline
\end{array}$$

The Manager Functions **SwitchM**, **SwitchU**, **TopUp** and **Collect** are defined as follows:

$$\begin{array}{|l}
\hline
\textit{SwitchM} \\
i? : BUTTONS \\
r! : \mathsf{Lights} \\
\hline
i? = \mathcal{M} \\
r! = \mathsf{ManageOn} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{SwitchU} \\
i? : BUTTONS \\
r! : \mathsf{Lights} \\
\hline
i? = \mathcal{U} \\
r! = \mathsf{ManageOff} \\
\hline
\end{array}$$

TopUp

$$
\begin{array}{|l}
i? : Coins \times \textsc{Slots} \\
\Delta State \\
r! : \mathbb{N} \\
\hline
i? = (c, \textsc{ManagerIn}) \\
Credit' = Credit \\
Bank' = Bank \oplus \{c \mapsto Bank(c) + 1\} \\
r! = Value(Bank)
\end{array}
$$

Collect

$$
\begin{array}{|l}
i? : BUTTONS \\
\Delta State \\
r! : ((Coins \rightarrow \mathbb{N}) \times slots) \\
\hline
i? = \mathcal{E} \\
Credit' = Credit \\
Bank' = \{(x, 0) \mid x \in Coins\} \\
r! = (Bank, \textsc{ManagerOut})
\end{array}
$$

It is worth noting that the operation *Change* is nondeterministic : any choice of coins, from *Bank*, of total value equal to *Credit* will suffice. The actual implementation of *Change* is, however, likely to be deterministic. Thus, in this case it is desirable to allow the specification to be quasi-nondeterministic but we should test for some weaker form of correctness than equivalence. This weaker form of correctness, that shall be called conformance, will be defined in the next section.

The input alphabet is the union of the input spaces of the operations. Similarly, the output alphabet is the union of the output spaces of the operations.

4.1. Conformance

When defining a formal specification language it is essential to state what it means for an implementation to be correct. Where a stream X-machine M is deterministic, an IUT I *conforms* to M if and only if it describes the same input/output function as M. Similarly, it is possible to define correctness to be equivalence when considering quasi-nondeterministic specifications. In this section we shall, however, define a weaker notion of conformance that shall be used throughout this paper.

It is quite normal, when defining conformance, to insist that:

- the implementation is defined where the specification is defined;
- for any input on which the specification is defined, any possible behaviour of the implementation is consistent with the specification.

The semantics defined earlier mean that M is completely specified. This behaviour is defined by $\lfloor M \rfloor_\perp$. Thus, $\lfloor M \rfloor_\perp$ and $\lfloor M_I \rfloor_\perp$ are defined on the same set on input strings and for M_I to conform to M we need $\lfloor M \rfloor$ and $\lfloor M_I \rfloor$ to have the same input domains.

IUT I conforms to quasi-nondeterministic stream X-machine M on input domain D if for every input sequence $\overline{x} \in D$, any output sequence $\overline{y}$ that I may produce when it receives input $\overline{x}$ satisfies $(\overline{x}, \overline{y}) \in \lfloor M \rfloor_\perp$. This shall be denoted $I \preceq_D M$. The following provides a more formal definition.

Definition 8. Given quasi-nondeterministic stream X-machine M, if the IUT I behaves like some quasi-nondeterministic stream X-machine M_I then $I \preceq_D M$ if and only if $\forall \overline{x} \in D.(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor_\perp \Rightarrow (\overline{x}, \overline{y}) \in \lfloor M \rfloor_\perp$.

It is now possible to formally define what it means for I to conform to M.

Definition 9. Given quasi-nondeterministic stream X-machine M, the IUT I *conforms* to M if and only if $I \preceq_{In^*} M$.

Assuming I behaves like a quasi-nondeterministic stream X-machine M_I, this is equivalent to $\lfloor M_I \rfloor_\perp \subseteq \lfloor M \rfloor_\perp$. In contrast, the approache given in ([17]) considers I to conform to M if and only if $\lfloor M_I \rfloor_\perp = \lfloor M \rfloor_\perp$. Throughout this paper $I \preceq M$ shall denote $I \preceq_{In^*} M$ and $I \not\preceq M$ shall denote $\neg (I \preceq M)$.

5. Testing against a deterministic stream X-machine

One of the great advantages of using deterministic stream X-machines in software development is the power of the associated test generation algorithm. The essential philosophy behind the *stream X-machine testing (SXMT) method* is that the implementation is built from a number of, possibly small, components that are believed to be correct. Testing then reduces to determining whether these components have been combined in the correct way. Associated with this philosophy is a notion of refinement under which these components may, themselves, have been tested using this method (see, for example, [16]).

The SXMT method is based on an FSM algorithm due to Chow ([4]) and is *guaranteed* to determine correctness as long as the specification and the IUT satisfy certain design for test conditions. These conditions shall now be described. The test generation algorithm will then be given.

5.1. Design for test conditions

Given a stream X-machine M, it is possible to abstract M to form a finite automaton, called the *associated automaton*, $A(M)$ by keeping the logical state structure, representing each function from Φ by a symbol and labelling each arc by the corresponding symbol ([15]). Tests are generated from $A(M)$ using the W-method.

Before the design for test conditions are stated two definitions from [15] will be given. The first of these uses the projection function π_i that takes a tuple and returns the i^{th} element of the tuple.

Definition 10. Φ is *output distinguishable* if $\forall f, f' \in \Phi, m \in Mem, x \in In$ such that $(m, x) \in dom\ f \cap dom\ f'$, $f \neq f' \Rightarrow \pi_1(f(m, x)) \neq \pi_1(f'(m, x))$.

This property means that two functions from Φ cannot produce the same output given input x and memory m. Output distinguishability allows tests to distinguish between different functions. The following condition says that given any memory value m and function f there is some input x that allows f to be triggered when the memory is m. This simplifies the problem of finding an input sequence to trigger a sequence of functions.

Definition 11. Φ is *complete*, with respect to Mem, if $\forall\, m \in Mem, f \in \Phi. \exists\, x \in In.(m, x) \in dom\ f$.

The following assumptions, called design for test conditions, are made in the SXMT method ([15])

Definition 12. Suppose implementation I is to be tested against stream X-machine M that has set Φ of processing functions. M and I satisfy the *design for test conditions* for the SXMT method if the following hold.

1. M is deterministic.
2. Φ is output distinguishable.
3. Φ is complete with respect to Mem.
4. I behaves like some deterministic minimal stream X-machine M_I that has the same input alphabet, output alphabet and set of processing functions as M.
5. There is a known integer n' such that M_I has at most n' states.

Where necessary, the design for test conditions for Φ may be satisfied by adding extra input symbols, that trigger the functions, and extra output symbols that allow the functions to be distinguished. These additional inputs and outputs might be removed after the test set generated by the SXMT method has been applied. Naturally, the IUT might then be retested in order to check that no new faults have been introduced. While the addition of these extra values may slightly increase the software development cost, this will often be justified by the assistance provided to testing.

It is possible to distinguish between conditions placed on Φ and conditions placed on I. The first condition given above simply limits the approach to deterministic specifications. The second and third conditions are properties of Φ only and thus may be achieved by the development of an appropriate specification. They shall thus be called *specify for test conditions*. The remaining two conditions are conditions on the IUT and will be called *test hypotheses* ([8]). Thus the set of design for test conditions is the union of the set of specify for test conditions, the set of test hypotheses and the determinism condition.

The fourth condition holds if the system is built from a number of components that are known to be correct. This property may be achieved by refining the system down to relatively small components in which there is a high degree of confidence or by building I from a set of trusted reused components ([16]). The fourth condition is an example of a type of hypothesis often used when deriving tests from a formal specification or model: that the IUT is contained within some set of models described in a particular formalism (see, for example, [19]).

The fifth condition requires the tester to apply expert knowledge. In some situations the choice of n' will be clear. Alternatively, the choice of n' may be based upon pragmatic factors ([4]).

5.2. Test generation

As noted above, it is normal to assume that, for some predefined n', I behaves like some unknown deterministic steam X-machine M_I that has at most n' states. As M and M_I are deterministic, I *conforms* to M if and only if M and M_I are equivalent. Assuming the specify for test conditions and test hypotheses hold, it is possible to adapt the *W-method*, that was originally introduced by Chow, for testing against a deterministic FSM ([4]).

Given a deterministic stream X-machine M let $A(M)$ denote the associated minimal automaton. Thus $A(M)$ has input alphabet Φ. A *state cover* V of $A(M)$ is a set of input sequences (from Φ^*) that, between them, reach every state of $A(M)$ exactly once. Thus, for each state s_i of $A(M)$ there is some unique sequence $\overline{v}_i \in V$ that takes $A(M)$ from its initial state to s_i. As $A(M)$ is minimal, it is initially connected and thus has a state cover. V may represent a spanning tree of $A(M)$ that is rooted at s_1.

Let n denote the number of states of M and n' denote an upper bound on the number of states of M_I. Further, let V be a state cover for $A(M)$ and W a characterizing set for $A(M)$. Then the following provides a set of sequences from $A(M)$ ([16]).

$$T_F = V(\{\epsilon\} \cup \Phi \cup \Phi^2 \cup \ldots \cup \Phi^{n'-n+1})\, W$$

Recall that $\langle \overline{f} \rangle$ denotes the set of input/output sequences that correspond to the execution of $\overline{f}$ from initial memory value m_0. The design for test condition that Φ is complete with respect to *Mem* guarantees that given $\overline{f} \in \Phi^*$ there is some input sequence in *dom* $\langle \overline{f} \rangle$.

It is sufficient to take each $\overline{f} \in T_F$, and generate a corresponding input sequence. This will be achieved by the application of a test function $t_{s,m}$ that takes a sequence $\overline{f}$ of functions and derives an input sequence that corresponds to executing $\overline{f}$ from state s and memory m. The sequence $t_{s,m}(\overline{f})$ will be used to determine whether $\overline{f}$ can be executed from s.

Definition 13. The *test function* $t_{s,m}$ is defined by the following ([18]).

1. $t_{s,m}(\epsilon) = \epsilon$.
2. If $\overline{f} \in L_{A(M)}(s)$, $t_{s,m}(\overline{f}) = \overline{x}_1$, $\|\overline{f}\|\,(m, \overline{x}_1) = (\overline{y}_1, m')$ and $(m', x) \in dom\ f$ then $t_{s,m}(\overline{f}f) = \overline{x}_1 x$.
3. If $\overline{f} \notin L_{A(M)}(s)$, $t_{s,m}(\overline{f}f) = t_{s,m}(\overline{f})$

In the second and third rules, $f \in \Phi$ and $\overline{f} \in \Phi^*$.

Given some $\overline{f} \in T_F$, there is no guarantee that $\overline{f} \in L(A(M))$. Suppose $\overline{f} \in T_F \setminus L(A(M))$, and $\overline{f} = \overline{f}_1 f_2 \overline{f}_3$ for some $f_2 \in \Phi$ and maximal $\overline{f}_1 \in L(A(M))$. In M, the triggering of $\overline{f}_1$ should generate output and then f_2 should lead to $\perp$. Thus, it is only necessary to input a string to demonstrate that $\overline{f}_1$ can be executed from the initial state of M_I and that f_2 cannot be executed from the state reached by this.

The set of input sequences, produced from T_F by t_{s_1,m_0}, is guaranteed to determine correctness if Φ satisfies the design for test conditions and I satisfies the test hypotheses ([16]). We shall see later that, while the W-method was originally produced for completely specified finite automata and finite state machines, it can be applied to stream X-machines that are not completely specified.

6. Testing against a quasi-nondeterministic stream X-machine: design for test conditions

In this section the design for test conditions, used when testing against a deterministic stream X-machine, will be generalized to the case where tests are being derived from a quasi-nondeterministic stream X-machine. The section also contains results that prove that these new conditions are a generalization of the design for test conditions assumed when testing from a deterministic stream X-machine.

With quasi-nondeterministic stream X-machines it is necessary to generalize the notion of output distinguishability, to nd-output distinguishability. Rather than require that any two functions defined on some (m, x) generate different output when executed with (m, x), it is necessary to require that the two sets of possible outputs are disjoint.

Definition 14. Φ is *nd-output distinguishable* if for all $f_1, f_2 \in \Phi$ with $f_1 \neq f_2$

$$(m, x) \in dom\ f_1 \cap dom\ f_2 \Rightarrow \{\pi_1(a) \mid a \in f_1(m, x)\} \cap \{\pi_1(a) \mid a \in f_2(m, x)\} = \varnothing$$

Where M is deterministic, one of the test hypotheses is that the function sets for M and M_I are the same. Here, instead, it is sufficient to assume that the relations from the relation set Φ' of M_I all conform to relations in Φ, where $f' \in \Phi'$ conforming to $f \in \Phi$ is defined by the following and is denoted $f' \leq f$.

Definition 15. Given $f' \in \Phi'$ and $f \in \Phi$, $f' \leq f$ if and only if $dom\ f' = dom\ f$ and $f' \subseteq f$.

Test generation may be complicated if, because of nondeterminism, after M_I has processed an input sequence, the expected memory is not known. This may make it difficult to find an input sequence that should trigger a particular sequence of transitions; at each stage the input chosen depends upon the memory. In order to eliminate this difficulty it will be assumed that given $f \in \Phi$, $m \in Mem$ and $x \in In$, if (y_1, m_1), $(y_2, m_2) \in f(m, x)$ then m_1 and m_2 may differ only if the outputs y_1 and y_2 also differ. This reduces the uncertainty associated with nondeterminism; by observing the output at each stage the expected memory may be determined. If Φ satisfies this condition it is said to be observable.

Definition 16. Φ is *observable* if and only if $\forall f \in \Phi, m \in Mem, x \in In.$

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

The specify for test conditions can now be given.

Definition 17. If Φ is the relation set of a quasi-nondeterministic stream X-machine with memory *Mem* then the *specify for test conditions* are

1. Φ is nd-output distinguishable;
2. Φ is complete with respect to *Mem*;
3. Φ is observable.

It is worth noting that, under the specify for test conditions, given $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$, there is exactly one $\overline{f} \in L(A(M))$ with $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$.

The following result shows that the specify for test conditions are a generalization of those used for deterministic stream X-machines.

Lemma 1. Suppose M is a deterministic stream X-machine with function set Φ. Φ satisfies the specify for test conditions for a deterministic stream X-machine if and only if Φ satisfies the specify for test conditions for a quasi-nondeterministic stream X-machine.

Proof

By definition, as all the elements of Φ are functions, Φ is observable. Clearly the definitions of complete with respect to *Mem* for deterministic and quasi-nondeterministic stream X-machines are equivalent when M is deterministic. Thus it is sufficient to prove that Φ is nd-output distinguishable if and only if Φ is output distinguishable.

Suppose $f_1, f_2 \in \Phi$, $m \in Mem$, $x \in In$, $f_1 \neq f_2$ and $(m,x) \in dom\ f_1 \cap dom\ f_2$. As M is deterministic f_1 and f_2 are both functions. Let $f_1(m,x) = (y_1, m_1)$ and $f_2(m,x) = (y_2, m_2)$. Thus

1. $\{y \mid \exists\, m'.((m,x),(y,m')) \in f_1\} = \{y_1\}$
2. $\{y \mid \exists\, m'.((m,x),(y,m')) \in f_2\} = \{y_2\}$.

There are now two cases to consider
Case 1: Φ is output distinguishable. Thus $y_1 \neq y_2$ and so $\{y \mid \exists\, m'.((m,x),(y,m')) \in f_1\} \cap \{y \mid \exists\, m'.((m,x),(y,m')) \in f_2\} = \varnothing$. Thus Φ is nd-output distinguishable as required.
Case 2: Φ is nd-output distinguishable. Thus $\{y \mid \exists\, m'.((m,x),(y,m')) \in f_1\} \cap \{y \mid \exists\, m'.((m,x),(y,m')) \in f_2\} = \varnothing$. From this, $y_1 \neq y_2$ and thus Φ is output distinguishable as required.

As in the deterministic case, it is possible to adapt Φ so that it satisfies the specify for test conditions. Where necessary, this may be achieved by extending the input and output domains. While this may increase the development cost these conditions lead to an automated test method that is as powerful as the SXMT. In many situations, the improved power and reduced cost of testing justifies any extra development costs.

The following extends the test hypotheses to the case where tests are being generated from a quasi-nondeterministic stream X-machine.

Definition 18. If M is a quasi-nondeterministic stream X-machine with set Φ of processing relations and I is the implementation to be tested against M then the *test hypotheses* are

1. I behaves like some (unknown) minimal quasi-nondeterministic stream X-machine M_I that does not have state nondeterminism, has the same input and output alphabets as M, and has a set Φ' of processing relations with the property that: for each $f' \in \Phi'$ there is some $f \in \Phi$ such that $f' \leq f$.
2. There is some known n' such that M_I has at most n' states.

Any quasi-nondeterministic stream X-machine can be rewritten to a minimal quasi-nondeterministic stream X-machine that does not have state nondeterminism and thus, if I behaves like a quasi-nondeterministic stream X-machines with set Φ' of processing relations it also behaves like a minimal quasi-nondeterministic stream X-machine that does not have state nondeterminism and has set Φ' of processing relations.

Implicit in the test hypotheses is the assumption that, in M_I, if two processing relations f and f' from Φ' can be executed from the same state of M_I then their input domains do not overlap. Naturally, this is *always* the case in the important case that I is deterministic.

The following result, which is an immediate consequence of the definitions of the test hypotheses, states that the test hypotheses given above are a generalization of the test hypotheses used when testing from a deterministic stream X-machine.

Lemma 2. Suppose M is a deterministic stream X-machine and I is an implementation being tested against M. I satisfies the test hypotheses used for deterministic stream X-machines if and only if I satisfies the test hypotheses used for quasi-nondeterministic stream X-machines.

By Lemmas 1 and 2, the design for test conditions used when testing from a quasi-nondeterministic stream X-machine are a generalization of those used when testing from a deterministic stream X-machine. The design for test conditions for quasi-nondeterministic stream X-machines also fit the philosophy of software development used with deterministic stream X-machines, in which the implementation is built from components that are believed to conform to components of the specification.

In order to illustrate the specify for test conditions for a quasi-nondeterministic stream X-machine we shall now show that the Vending Machine example satisfies these conditions.

6.1. Specify for test conditions and the Vending Machine

6.1.1. nd-Output Distinguishable

The functions are trivially nd-output distinguishable, because their ranges are distinct.

6.1.2. Completeness

To be complete, for every possible memory, a relation must have at least one input which triggers the function.

For `Insert` every input triggers the relation in every input state. For `Vend`, there is only one possible input $\mathcal{V}$. This clearly triggers `Vend` in every possible memory state because the antecedents of the two implications which define it are mutually exhaustive. For `Change`, there is also only one input possible: $\mathcal{C}$. This always triggers the relation, since the change returned can be 'empty'. For `SwitchM` there is only one input, $\mathcal{M}$, which always triggers the relation. For `SwitchU` there is only one input, $\mathcal{U}$, which always triggers the relation. For `TopUp`, as for `Insert`, every input triggers the relation in every state. Finally, for `Collect`, there is only one input, $\mathcal{E}$, which triggers the relation in every state.

6.1.3. Observability

All relations except `change` are deterministic and are therefore observable. The relation `change` is observable because all cases where change returns a different memory state (i.e. a different value for the 'bank' component of the state), it must also produce a different set of change.

7. Testing against a quasi-nondeterministic stream X-machine

In this section the W-method shall be applied to the problem of testing against a quasi-nondeterministic stream X-machine M. As is normal, it will be assumed that all states are final states.

In common with the deterministic case, the first step involves generating the associated automaton $A(M)$. As $A(M)$ may be nondeterministic and the W-method is applied to deterministic systems, the next step it to generate the minimal deterministic automaton $D(M)$ that is equivalent to $A(M)$. Tests shall be derived from $D(M)$.

Test generation may thus be divided into the following steps.

1. $A(M)$ is produced.
2. $D(M)$ is generated from $A(M)$.
3. State cover V and characterizing set W are derived for $D(M)$.
4. Chow's method is used to generate a set of sequences $T_F = V(\{\epsilon\} \cup \Phi \cup \Phi^2 \cup \ldots \cup \Phi^{n'-n+1})W$ where n is the number of states of $D(M)$ and n' is an upper bound on the number of states of M_I.
5. I is tested with test sequences that correspond to the elements of T_F.

The set of input sequences generated by this process is the test set T_X.

Due to nondeterminism, the process that produces tests must be adaptive; each input is chosen once the output, in response to the previous input, is received. Thus, we shall call it a *test process*. While there may be more than one possible memory after the processing of a value, due to Φ being observable the expected memory is known once the output has been observed. Thus, given $\overline{f}$ it is possible to develop a test associated with $\overline{f}$. This test has an input sequence $\overline{x}$ and an output sequence $\overline{y}$ observed in the generation of $\overline{x}$. The adaptive nature of test generation means that the input and output sequences should not be separated: the test process returns a pair.

The following defines the test process for quasi-nondeterministic stream X-machines.

Definition 19. The test process t^{nd} for a quasi-nondeterministic stream X-machine M that satisfies the specify for test conditions is defined by the following.

1. $t^{nd}(\epsilon) = (\epsilon, \epsilon)$.
2. Suppose $\overline{f} \in L(D(M))$ and $t^{nd}(\overline{f}) = (\overline{x}_1, \overline{y}_1)$. If $((m_0, \overline{x}_1), (\overline{y}_1, m')) \in \|\overline{f}\|$, $(m', x) \in dom\ f$ and I produces output y in response to the execution of x after $\overline{x}_1$ then $t^{nd}(\overline{f}f) = (\overline{x}_1 x, \overline{y}_1 y)$.
3. If $\overline{f} \notin L(D(M))$, $t^{nd}(\overline{f}f) = t^{nd}(\overline{f})$

Where $\overline{f}$ is not in $L(D(M))$ the last output, when the test function is applied, should be $\perp$.

Interestingly, while the W-method was designed to test from completely specified finite automata and finite state machines, it may be applied to a quasi-nondetereministic stream X-machine M that is not completely specified. In order to see this, it is sufficient to consider what happens if we completed M, to form M', by adding an error state. The test set generated might be extended for the following two reasons.

- Adding one element to V to reach the error state.
- Possibly extending W to distinguish this new state from the states of M.

Suppose T denotes the set of sequences of operations generated, using the W-method, from M and T' denotes the corresponding set generated from M' and that the state cover and characterizing sets for M' are developed by extending those of M. Clearly, T is strictly contained in T'. We shall see, however, that it is sufficient to choose $T = T'$.

Consider, initially, the extension of V to form V'. It is important to recall that an error, generated by the input of a value for which there is no defined behaviour, leads to termination. Thus, it is never necessary to consider input beyond one that should lead to an error. Suppose V is extended by some sequence $\overline{err} \in T$ that reaches the error state. Consider some test $\overline{t} \in T'$ such that $\overline{t}$ is $\overline{err}$ followed by $\overline{t'}$ (for some $\overline{t'}$). Then the test process will produce the same test sequence for $\overline{t}$ as for $\overline{err}$. In fact, all extensions of $\overline{err}$ in T' will, in effect, be reduced to $\overline{err}$ (which is in T) by the test process. Thus, as $\overline{t}$ is in T, it is not necessary to extend V.

Now, consider the extension to W. Since an error leads to immediate termination of the IUT, non-termination of the system indicates that we are not in the error state. Thus, it is not necessary to extend W in order to distinguish the error state from other states of the IUT.

To conclude, the test T generated from a quasi-nondeterministic stream X-machine M that is not completely specified is one that can be produced by applying the W-method to the quasi-nondeterministic stream X-machine produced by completing M.

7.1. Test generation for the Vending Machine

It will be assumed that the IUT has no more states than the specification of the vending machine and thus that $n' = n$.

The state cover, V needs to reach two states, and therefore contains two sequences. The empty sequence takes the system to the initial state and the sequence $\langle \texttt{switchM} \rangle$ takes it from the initial state to the Managing state.

$V = \{\epsilon, \langle \texttt{switchM} \rangle\}$

The characterizing set must contain a sequence of inputs which distinguishes every pair of states. As there are only two states in the Vending Machine, there need be only a single sequence in the characterizing set:

$W = \{\langle \texttt{switchM} \rangle\}$

The set Φ, contains the edge labellings for each transition in the associated deterministic finite automaton.

$\Phi = \{\texttt{Change}, \texttt{Vend}, \texttt{Insert}, \texttt{switchM}, \texttt{switchU}, \texttt{Collect}, \texttt{TopUp}\}$

The sequences of functions which the test cases must execute, T_F are thus:

$$\begin{array}{rcl}
T_F & = & V(\{\epsilon\} \cup \Phi) W \\
& = & VW \cup V\Phi W \\
& = & VW \cup \Phi\{\langle\texttt{switchM}\rangle\} \cup \{\langle\texttt{switchM}\rangle\}\Phi\{\langle\texttt{switchM}\rangle\} \\
& = & \{\langle\texttt{switchM}\rangle, \langle\texttt{switchM}, \texttt{switchM}\rangle\} \cup \\
& & \{\langle\texttt{change}, \texttt{switchM}\rangle, \langle\texttt{vend}, \texttt{switchM}\rangle, \langle\texttt{insert}, \texttt{switchM}\rangle, \\
& & \langle\texttt{switchM}, \texttt{switchM}\rangle, \langle\texttt{switchU}, \texttt{switchM}\rangle, \langle\texttt{collect}, \texttt{switchM}\rangle, \langle\texttt{TopUp}, \texttt{switchM}\rangle\} \cup \\
& & \langle\texttt{switchM}\rangle\Phi\langle\texttt{switchM}\rangle \\
& = & \{\langle\texttt{switchM}\rangle, \langle\texttt{change}, \texttt{switchM}\rangle, \langle\texttt{vend}, \texttt{switchM}\rangle, \langle\texttt{insert}, \texttt{switchM}\rangle, \\
& & \langle\texttt{switchM}, \texttt{switchM}\rangle, \langle\texttt{switchU}, \texttt{switchM}\rangle, \langle\texttt{collect}, \texttt{switchM}\rangle, \langle\texttt{TopUp}, \texttt{switchM}\rangle, \\
& & \langle\texttt{switchM}, \texttt{change}, \texttt{switchM}\rangle, \langle\texttt{switchM}, \texttt{vend}, \texttt{switchM}\rangle, \langle\texttt{switchM}, \texttt{insert}, \texttt{switchM}\rangle, \\
& & \langle\texttt{switchM}, \texttt{switchM}, \texttt{switchM}\rangle, \langle\texttt{switchM}, \texttt{switchU}, \texttt{switchM}\rangle, \\
& & \langle\texttt{switchM}, \texttt{collect}, \texttt{switchM}\rangle, \langle\texttt{switchM}, \texttt{TopUp}, \texttt{switchM}\rangle\}
\end{array}$$

In order to determine the input and output associated with a the test process t^{nd} is applied. To illustrate, suppose $t^{nd}(\langle\texttt{switchM}\rangle)$ is to be calculated:

$$\begin{array}{rcl}
t^{nd}(\langle\texttt{switchM}\rangle) & = & t^{nd}(\epsilon\langle\texttt{switchM}\rangle) \\
& = & (\overline{x}_1 x, \overline{y}_1 y) \\
\text{where} & & (\overline{x}_1, \overline{y}_1) = t^{nd}(\epsilon) = (\epsilon, \epsilon) \\
\text{and} & & ((m_0, \overline{x}_1), (\overline{y}_1, m')) \in \|\epsilon\| \\
\text{and} & & (m', x) \in \operatorname{dom} \texttt{switchM} \\
\\
\Rightarrow & & m' = m_0 \wedge x = \mathcal{M} \wedge y = \mathsf{ManageOn} \\
\\
\text{So} & & t^{nd}(\langle\texttt{switchM}\rangle) = (\mathcal{M}, \mathsf{ManageOn})
\end{array}$$

Where the sequence to be triggered is of the form $\overline{f}f$ for some $\overline{f}$ that is not in the language of the associated automaton the test process employs the third clause of the definition of t^{nd} to reduce the length of the sequence to be triggered. For instance,

$$\begin{array}{c}
t^{nd}(\langle\texttt{switchM}, \texttt{change}, \texttt{switchM}\rangle) \\
= \\
t^{nd}(\langle\texttt{switchM}, \texttt{change}\rangle)
\end{array}$$

Thus, the input sequence $\langle\mathcal{M}, \mathcal{C}\rangle$ is produced and the expected output is $\langle\mathsf{ManageOn}, \perp\rangle$.

Using the test process, t^{nd}, the test input required to trigger $\langle\texttt{change}, \texttt{switchM}\rangle$ can be calculated to be $\langle\mathcal{C}, \mathcal{M}\rangle$. The output sequence which must be observed for this input sequence is calculated (using t^{nd}) to be $\langle(0, \textsc{UserOut}), \mathsf{ManageOn}\rangle$. That is, no change should appear at the user output slot and the 'Manage on' light should illuminate for 2 seconds.

8. Proof that the test determines correctness

Throughout this section it will be assumed that $M = (In, Out, S, Mem, \Phi, F, s_0, m_0, T)$ is a quasi-nondeterministic stream X-machine with n states that satisfies the specify for test conditions, for quasi-nondeterministic stream X-machines, and in which every state is a final state. It shall also be assumed that I is an IUT that satisfies the test hypotheses for quasi-nondeterministic stream X-machines with M_I, Φ' and n' as defined earlier.

This section contains a proof that, under the specify for test conditions and test hypotheses, the test generated by the algorithm given in Section 7 is *guaranteed* to determine correctness.

The relation $\leq$ can be extended to $\leq^*$ by applying the following rules.

- $\epsilon \leq^* \epsilon$.
- $\forall f \in \Phi, \overline{f} \in \Phi^*, f' \in \Phi', \overline{f'} \in \Phi'^*. f'\overline{f'} \leq^* f\overline{f} \Leftrightarrow f' \leq f \wedge \overline{f'} \leq^* \overline{f}$.

Testing may be seen as a process of comparing $D(M)$ and $A(M_I)$. This comparison is complicated by these automata having different alphabets, Φ and Φ'. In order to simplify this comparison an abstraction $Abs(M_I)$ of $A(M_I)$ will be defined. $Abs(M_I)$ is the FA produced by replacing each label f' of $A(M_I)$ by the corresponding $f \in \Phi$ which satisfies $f' \leq f$. $Abs(M_I)$ shall be called the Φ-*abstraction* of M_I. Similarly, given $f' \in \Phi$, $abs(f')$ shall denote the element f of Φ' that satisfies $f' \leq f$. The first two results in this section demonstrate that $Abs(M_I)$ and $abs(f')$ are uniquely defined.

It will transpire that M_I conforms to M if and only if $Abs(M_I)$ and $D(M)$ are equivalent. In order to prove this, two results that relate sequences of functions executed in $D(M)$ and $A(M_I)$, are given. We then prove that $I \preceq M$ if and only if $L(Abs(M_I)) = L(D(M))$. These are followed by a proof that, given $\overline{f} \in \Phi^*$, if $(\overline{x}, \overline{y}) = t^{nd}(\overline{f})$ and $(\overline{x}, \overline{y}) \in \lfloor M \rfloor_\perp$ then either $\overline{f}$ is in both $L(D(M))$ and $L(Abs(M_I))$ or it is in neither of these languages. The section concludes with a proof that the test generated from T_F by t^{nd} determines whether $D(M)$ and $Abs(M_I)$ are equivalent and thus whether I conforms to M.

The following results demonstrate that, for each f' there is exactly one $f \in \Phi$ with $f' \leq f$ and thus that $abs(f')$ and $Abs(M_I)$ are uniquely defined.

Lemma 3. Suppose $\overline{f}, \overline{g}$ are non-empty sequences from Φ^* and $\langle \overline{f} \rangle \cap \langle \overline{g} \rangle \neq \varnothing$. Then $\overline{f} = \overline{g}$.

Proof

Proof by induction on $\overline{f}$. Since Φ is nd-output distinguishable, the result clearly holds for the base case where $\overline{f}$ has length 1.
Suppose that $\overline{f}$ has length greater than 1. Clearly as $\langle \overline{f} \rangle \cap \langle \overline{g} \rangle \neq \varnothing$, $\overline{f}$ and $\overline{g}$ the same length. Then $\overline{f} = \overline{f}_1 f_2$ and $\overline{g} = \overline{g}_1 g_2$ for some $f_2, g_2 \in \Phi$.
By the inductive hypothesis, $\overline{f}_1 = \overline{g}_1$. Suppose $(\overline{x}_1 x_2, \overline{y}_1 y_2) \in \langle \overline{f} \rangle \cap \langle \overline{g} \rangle = \langle \overline{f}_1 f_2 \rangle \cap \langle \overline{f}_1 g_2 \rangle$ ($x_2 \in In$, $y_2 \in Out$). Since Φ is observable there is some unique memory m with $(\overline{y}_1, m) \in \|\overline{f}_1\|$. Then, both f_2 and g_2 can produce output y_2 in response to input x_2 when in memory m. Thus, since Φ is nd-output distinguishable, $f_2 = g_2$ and so $\overline{f} = \overline{g}$ as required.

Lemma 4. Suppose $\overline{f'} \in \Phi'^*$. Then there is exactly one sequence $\overline{f}$ in Φ^* with $\overline{f'} \leq^* \overline{f}$.

Proof

By the definition of Φ^*, there must always be some $\overline{f} \in \Phi^*$ with $\overline{f'} \leq^* \overline{f}$. It is now sufficient to consider uniqueness. The case where $\overline{f'}$ is empty clearly holds and the case where $\overline{f'}$ is non-empty follows directly from Lemma 3.

The following result relates $\overline{f}$ to the sequence triggered in M_I by $t^{nd}(\overline{f})$ if $t^{nd}(\overline{f}) \in \lfloor M \rfloor$.

Lemma 5. Suppose t^{nd} is the test process, $\overline{f} \in L(D(M))$ and $(\overline{x}, \overline{y}) = t^{nd}(\overline{f})$. If $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$ then the sequence $\overline{f'} \in \Phi'^*$ triggered when M_I produces $\overline{y}$ in response to $\overline{x}$ satisfies $\overline{f'} \leq^* \overline{f}$.

Proof

The result clearly holds when $\overline{f'}$ is of length 0. Suppose now that $\overline{f'}$ has length greater than 0. By Lemma 4 there is some unique $\overline{f}_1 \in \Phi^*$ with $\overline{f'} \leq^* \overline{f}_1$. Thus, by the definition of $\leq^*$, $\overline{f}_1$ is capable of producing output $\overline{y}$ in response to input $\overline{x}$. Further, by the definition of t^{nd}, $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$. Thus $\langle \overline{f} \rangle \cap \langle \overline{f}_1 \rangle \neq \varnothing$ and by Lemma 3 $\overline{f} = \overline{f}_1$. Thus, $\overline{f'} \leq^* \overline{f}$ as required.

The following result shows that if I conforms to M then the sequences from $L(A(M_I))$ correspond to sequences from $L(D(M))$.

Lemma 6. Suppose that $I \preceq M$ and $\overline{f'} \in L(A(M_I))$. Then there is some sequence $\overline{f} \in L(D(M))$ with $\overline{f'} \leq^* \overline{f}$.

Proof

The result clearly holds when $\overline{f'}$ is of length 0. Consider now the cases where $\overline{f'}$ is of length greater than 0. Suppose $(\overline{x}, \overline{y}) \in \langle \overline{f'} \rangle$. By Lemma 4, there is some unique $\overline{f}_1 \in \Phi^*$ with $\overline{f'} \leq^* \overline{f}_1$. As $\overline{f'} \in L(A(M_I))$, $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor$. Further, since $I \preceq M$, $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$ and thus there is some $\overline{f} \in L(D(M))$ with $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$. Thus, as $(\overline{x}, \overline{y}) \in \langle \overline{f} \rangle$ and $(\overline{x}, \overline{y}) \in \langle \overline{f}_1 \rangle$, so by Lemma 3, $\overline{f} = \overline{f}_1$. Therefore $\overline{f'} \leq^* \overline{f}_1$ and $\overline{f}_1 = \overline{f} \in L(D(M))$ and thus the result follows.

The following results demonstrate that I conforms to M if and only if $Abs(M_I)$ is equivalent to $D(M)$. They thus reduce the problem of deciding whether I conforms to M to determining whether $Abs(M_I)$ and $D(M)$ are equivalent.

Lemma 7. If $L(Abs(M_I)) = L(D(M))$ then $I \preceq M$.

Proof

Proof by contradiction will be used, assuming that $L(Abs(M_I)) = L(D(M))$ but that $I \not\preceq M$.
As $I \not\preceq M$, $\lfloor M_I \rfloor_\perp \not\subseteq \lfloor M \rfloor_\perp$. Thus there is some minimal length $\overline{x}$ such that there is some $\overline{y}$ with $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor_\perp \setminus \lfloor M \rfloor_\perp$. Clearly $\overline{x}$ has length at least 1. There are two cases to consider: $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor$ and $(\overline{x}, \overline{y}) \notin \lfloor M_I \rfloor$
Case 1: $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor$. Then there is some $\overline{f'} \in L(M_I)$ such that $(\overline{x}, \overline{y}) \in \langle \overline{f'} \rangle$. Thus, as $L(Abs(M_I)) = L(D(M))$, there is some $\overline{f} \in L(D(M))$ with $\overline{f'} \leq^* \overline{f}$. Thus, $(\overline{x}, \overline{y}) \in \langle f \rangle \subseteq \lfloor M \rfloor$ providing a contradiction as required.
Case 2: $(\overline{x}, \overline{y}) \notin \lfloor M_I \rfloor$. Let $\overline{x} = \overline{x}_1 x_2$ and $\overline{y} = \overline{y}_1 y_2$, some $x_2 \in In$ and $y_2 \in Out \cup \{\perp\}$. By the minimality of $\overline{x}$, $(\overline{x}_1, \overline{y}_1) \in \lfloor M_I \rfloor_\perp \cap \lfloor M \rfloor_\perp$. Here we may note that since $\perp$ denotes halting, if $\overline{y}_1$ included $\perp$, the response of M_I and M to further input would be the same (null). Thus $\overline{y}_1$ does not contain $\perp$ and so $(\overline{x}_1, \overline{y}_1) \in \lfloor M_I \rfloor \cap \lfloor M \rfloor$.
Since $(\overline{x}, \overline{y}) \notin \lfloor M_I \rfloor$, $y_2 = \perp$. Since $(\overline{x}_1, \overline{y}_1) \in \lfloor M \rfloor$ and $(\overline{x}_1 x_2, \overline{y}_1 \perp) \notin \lfloor M \rfloor_\perp$, there must be some $y_3 \in Out$ and some $\overline{f} \in L(D(M))$ such that $(\overline{x}, \overline{y}_1 y_3) \in \langle \overline{f} \rangle$. As $L(D(M)) = L(Abs(M_I))$ there is some $\overline{f'} \in L(Abs(M_I))$ such that $\overline{f'} \leq^* \overline{f}$. Then, by the definition of $\leq^*$, $\overline{f'}$ and f have the same input domains, contradicting $y_2 = \perp$ as required.

Lemma 8. If $I \preceq M$ then $L(Abs(M_I)) = L(D(M))$.

Proof

Proof by contradiction will again be used, assuming that $I \preceq M$ but $L(Abs(M_I)) \neq L(D(M))$. There are thus two cases: there exists some $\overline{f}_0 \in L(Abs(M_I)) \setminus L(D(M))$ or there is some $\overline{f}_0 \in L(D(M)) \setminus L(Abs(M_I))$.
Case 1: $\exists \overline{f}_0 \in L(Abs(M_I)) \setminus L(D(M))$. As $\overline{f}_0 \in L(Abs(M_I))$ there is some sequence $\overline{f'} \in L(M_I)$ with $\overline{f}_0 = abs(\overline{f'})$. Choose $(\overline{x}, \overline{y}) \in \langle \overline{f'} \rangle$. By Lemma 6, there is some $\overline{f} \in L(D(M))$ with $\overline{f'} \leq^* \overline{f}$. As $\overline{f'} \leq \overline{f}_0$, $\overline{f'} \leq^* \overline{f}$, by Lemma 4 $\overline{f}_0 = \overline{f'}$. Thus, $\overline{f}_0 \in L(D(M))$, providing a contradiction as required.
Case 2: $\exists \overline{f} \in L(D(M)) \setminus L(Abs(M_I))$. Let $(\overline{x}, \overline{y}) = t^{nd}(\overline{f})$. As $I \preceq M$, by the definition of t^{nd} we know that $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$ and $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor$. Thus there is some $\overline{f'} \in L(A(M_I))$ such that $(\overline{x}, \overline{y}) \in \langle \overline{f'} \rangle$. By Lemma 5, $\overline{f'} \leq \overline{f}$ and thus $abs(\overline{f'}) = \overline{f}$. Thus $\overline{f} \in L(Abs(M_I))$, contradicting the choice of $\overline{f}$ as required.

Theorem 9. $I \preceq M$ if and only if $Abs(M_I)$ is equivalent to $D(M)$.

Proof

This follows directly from Lemmas 7 and 8.

Thus, in order to determine whether I conforms to M, it is sufficient to determine whether $Abs(M_I)$ and $D(M)$ are equivalent. In order to prove that the test process applied to T_F establishes the equivalence of $Abs(M_I)$ and $D(M)$ it is sufficient to initially prove that if $\overline{f} \in \Phi^*$ and $(\overline{x}, \overline{y}) = t^{nd}(\overline{f})$. From $(\overline{x}, \overline{y}) \in \lfloor M_I \rfloor_\perp$ it is then possible to deduce that $Abs(M_I)$ and $D(M)$ agree on $\overline{f}$: either both automata accept $\overline{f}$ or neither does.

Theorem 10. Suppose $\overline{f} \in \Phi^*$ and $t^{nd}(\overline{f}) = (\overline{x}, \overline{y})$. If $(\overline{x}, \overline{y}) \in \lfloor M \rfloor_\perp$ then either $\overline{f}$ is in both $L(Abs(M_I))$ and $L(D(M))$ or $\overline{f}$ is in neither $L(Abs(M_I))$ nor $L(D(M))$.

Proof

There are two cases: $\overline{f} \in L(D(M))$ and $\overline{f} \notin L(D(M))$.
Case 1: $\overline{f} \in L(D(M))$. Then $(\overline{x}, \overline{y}) \in \langle\overline{f}\rangle$ and $(\overline{x}, \overline{y}) \in \langle\overline{f'}\rangle$ for some $\overline{f'}$ from $L(A(M_I))$. By Lemma 5, $\overline{f'} \leq^* \overline{f}$. Thus, as $abs(\overline{f'}) = \overline{f}$, $\overline{f} \in L(Abs(M_I))$ as required.
Case 2: $\overline{f} \notin L(D(M))$. Proof by contradiction shall be used, assuming that $\overline{f} \in L(Abs(M_I))$. Since $\overline{f} \in L(Abs(M_I))$ there is some unique $\overline{f'} \in L(M_I)$ such that $\overline{f'} \leq^* \overline{f}$. Then, by the construction applied in t^{nd}, $(\overline{x}, \overline{y}) \in \langle\overline{f'}\rangle$ and thus $\overline{y}$ does not contain $\perp$. Thus $(\overline{x}, \overline{y}) \in \lfloor M \rfloor$ and so there is some $\overline{f}_1 \in L(D(M))$ with $(\overline{x}, \overline{y}) \in \langle\overline{f}_1\rangle$. But $\overline{f'} \leq^* \overline{f}$ and so $(\overline{x}, \overline{y}) \in \langle\overline{f}\rangle$ and thus, by Lemma 3, $\overline{f}_1 = \overline{f}$ and so $\overline{f} \in L(D(M))$. This provides a contradiction as required.

The above result can easily be generalized to sets of sequences from Φ^*, as shown below.

Theorem 11. Suppose P is a set of sequences from Φ and $P_t = \{t^{nd}(\overline{f}) \mid \overline{f} \in P\}$. If $P_t \subseteq \lfloor M \rfloor_\perp$ then for each $\overline{f} \in P$ either $\overline{f}$ is in both $L(Abs(M_I))$ and $L(D(M))$ or $\overline{f}$ is not in either $L(Abs(M_I))$ or $L(D(M))$.

Proof

Proof by induction on $\mid P \mid$. The result clearly holds for the base case $P = \{\}$.
Suppose $\overline{f} \in P$ and $P' = P \setminus \{\overline{f}\}$. Then, by the inductive hypothesis, $Abs(M_I)$ and $D(M)$ agree on P'. But, by Theorem 10, as $t^{nd}(\overline{f}) \in \lfloor M \rfloor_\perp$, $Abs(M_I)$ and $D(M)$ agree on $\overline{f}$. Thus $Abs(M_I)$ and $D(M)$ agree on $P' \cup \{\overline{f}\} = P$ as required.

The following demonstrates that it is sufficient to apply the W-method to generate a test set from $D(M)$.

Theorem 12. Suppose M is a complete, nd-output distinguishable, observable quasi-nondeterministic stream X-machine with n states and relation set Φ. Suppose also that IUT I behaves like some unknown minimal quasi-nondeterministic stream X-machine M_I without state nondeterminism that has at most n' states and relation set Φ' that satisfies: $\forall f' \in \Phi' \, \exists f \in \Phi . f' \leq f$. Let t^{nd} denote the test process and T_F denote the set of sequences of relations from Φ^* generated using the W-method. Let $T'_X = \{t^{nd}(\overline{f}) \mid \overline{f} \in T_F\}$. Then $I \preceq M$ if and only if $T'_X \subseteq \lfloor M \rfloor_\perp$.

Proof

From Theorem 9, $I \preceq M$ if and only if $Abs(M_I)$ is equivalent to $D(M)$. By [4], $Abs(M_I)$ is equivalent to $D(M)$ if and only if they agree on T_F. By Theorem 11, $Abs(M_I)$ and $D(M)$ agree on T_F if and only if $T'_X \subseteq \lfloor M \rfloor_\perp$. The result thus follows.

9. Future work

A number of algorithms, for testing against a deterministic finite state machine, have been introduced (see, for example, [7, 24, 25]). Under certain conditions, these are guaranteed to produce shorter test sequences that the W-method. One piece of future work is to investigate whether any of these alternative algorithms may be used when testing from a quasi-nondeterministic stream X-machine.

Two types on nondeterminism have been considered in this paper: state nondeterminism and operator nondeterminism. A third form of nondeterminism may be obtained by allowing operations that may be applied at a state s to have overlapping input domains. This involves removing the restriction that for all $s \in S$ and $f, f' \in \Phi$ such that $(s, f), (s, f') \in dom\ F, f \neq f' \Rightarrow dom\ f \cap dom\ f' = \{\}$. It would be interesting to extend the test generation algorithm given in this paper to allow this form of nondeterminism.

10. Conclusions

Deterministic stream X-machines have been used to specify software systems. One of the great benefits of using deterministic stream X-machines is the existence of a test method that, under certain well defined conditions, is guaranteed to determine correctness. Nondeterminism aids abstraction and is thus an attractive tool in the formulation of specifications. The restriction to deterministic specifications is therefore unhelpful and prohibits the application of the powerful stream X-machine method in cases where the specification is nondeterministic. Some implementations may also be nondeterministic or appear to be nondeterministic at the level of abstraction being considered.

Two ways of introducing nondeterminism into stream X-machines have been described: allowing more than one possible next state after an operation f has been performed from a state s and allowing the operations to be relations rather than functions. Interestingly operations are not allowed to be relations under the traditional definition of a stream X-machine. For this reason, this paper generalized the definition of a stream X-machine. A stream X-machine with these forms of nondeterminism has been called a quasi-nondeterministic stream X-machine.

This paper considered the problem of testing an implementation for conformance to a quasi-nondeterministic stream X-machine. The design for test conditions, used when testing from a deterministic stream X-machine, have been generalized to allow nondeterminism. An adaptive test generation algorithm has been given. As with the deterministic paradigm, the test generated by this algorithm is guaranteed to determine correctness under the design for test conditions.

Other authors have recently considered the problem of generating tests from a nondeterministic stream X-machine ([17]). However, they test for *equivalence* rather than *conformance*. In some cases conformance will be sufficient and will then be a more appropriate definition of correctness. The notion of correctness used would, naturally, depend upon the problem domain.

References

[1] 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.

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

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

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

[5] 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.

[6] J. H. Fetzer. Program verification: The very idea. *Communications of The ACM*, 31:1048–1063, 1988.

[7] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17:591–603, 1991.

[8] M. C. Gaudel. Testing can be formal too. In *TAPSOFT'95*, pages 82–96. Springer-Verlag, March 1995.

[9] A. Gill. *Introduction to The Theory of Finite State Machines.* McGraw-Hill, 1962.

[10] R. M. Hierons. Testing from a finite state machine: Extending invertibility to sequences. *The Computer Journal*, 40:220–230, 1997.

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

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

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

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

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

[16] M. Holcombe and F. Ipate. A method for refining and testing generalised machine specifications. *International Journal of Comp Math.*, 68:197–219, 1998.

[17] F. Ipate and M. Holcombe. Generating test sequences from non-deterministic X-machines. *Formal Aspects of Computing*, to appear.

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

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

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

[21] N. Mukerjee and T. J. Chakraborty. Built-in self-test: A complete test solution for telecommunications systems. *IEEE Communications Magazine*, 37:72–78, 1999.

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

[23] D. Rayner. Future directions for protocol testing, learning the lessons from the past. In *Testing of Communicating Systems: IFIT TC6 10th International Workshop on Testing of Communicating Systems*, pages 3–17, Cheju Island, Korea, September 1997. Chapman and Hall.

[24] A. Rezaki and H. Ural. Construction of checking sequences based on characterization sets. *Computer Communications*, 18:911–920, 1995.

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