26/3/98

Chapter 7.

Complete functional testing.

Summary. State machine morphisms, state machine minimality, state equivalence. State machine testing theory, test set constructions, transition cover, characterisation set, complexity. Stream X-machine testing, design for test conditions, fundamental test function, fundamental theorem of testing, expected outputs, test set complexity.

This chapter introduces the theoretical basis of the stream X-machine based testing (SXMT) method. This generates test sets for a system specified as a stream X-machine whose application ensures that the system behaviour is identical to that of the specification *provided* that the system is made of fault-free components and some explicit "design for testing" requirements are met. The method was applied to examples in Chapters 3 and 5.

7.1. Some state machine theory.

Some knowledge of the theory of finite state machines will be needed in our theoretical discussion and this section will present briefly several key concepts and results. We will refer to deterministic *finite state machines without outputs* (*automata*) as defined in the first chapter of the book, denoted by tuples of the form $\mathcal{A} = (Input, Q, \mathbf{F}, q_0)$, where *Input* is the input alphabet, Q the state set, q_0 is the initial state and **F** the next-state (partial) function. As in the previous chapter we shall assume that all the machine's states are terminal states, i.e. $T = Q$. The concepts and results given here are also valid for a (possibly infinite) state machine. Most of the results are well known, see for example Eilenberg [28], and we will not prove them here.

Morphisms.
A *morphism* is a particular type of state mapping between two state machines. We will need to consider the mathematical relationships between two machines, one representing the specification and the other the implementation since we want to establish, as far as possible, that their behaviours are the same. A morphism is a means of mapping states from one machine to states of the other in a way that respects the machine structure of both.

Definition 7.1.1.
Let $\mathcal{A} = (Input, Q, \mathbf{F}, q_0)$ and $\mathcal{A}' = (Input, Q', \mathbf{F}', q_0')$ be two state machines *over the same input alphabet*. Then $g: \mathcal{A} \to \mathcal{A}'$ is called a *morphism* if $g: Q \to Q'$ is a function that satisfies the following.

1. $g(q_0) = q_0'$
2. $\forall\, q \in Q, input \in Input,\ g(\mathbf{F}(q, input)) = \mathbf{F}'(g(q), input)$

Thus the two initial states must be related and a transition in the first machine must relate to the transition of the related states in the second.

The second requirement is equivalent to the following.

2'. $\forall\, q \in Q, input \in Input$, ($\exists$ $input$: $q \to \theta$ an arc in $\mathcal{A}$) $\Leftrightarrow$ ($\exists$ $input$: $g(q) \to g(\theta)$ an arc in $\mathcal{A}'$).

If $g: Q \to Q'$ is a surjective morphism then $\mathcal{A}'$ is obtained from $\mathcal{A}$ by merging all states whose image through g is the same. If g is bijective then $\mathcal{A}$ and $\mathcal{A}'$ are identical up to a renaming of the state space. In this case g is called a *state machine isomorphism*.

Definition 7.1.2.
A bijective state machine morphism is called an *isomorphism*.

Lemma 7.1.1.
If g: $A \rightarrow A'$ is a morphism then A and A' accept the same language.

The *language* accepted by an automaton is the set of input sequences corresponding to paths in the machine.

State machine minimality.
This section defines the minimal machine of an automaton and shows that this is unique up to a relabelling of the state space. Minimal machines are machines with as few states as possible for a given behaviour.

Definition 7.1.3.
Let $A = (Input, Q, F, q_0)$ be a state machine. Then a state $q \in Q$ is called *accessible* if $\exists$ *input**: $q_0 \rightarrow q$ a path from the initial state q_0 to q. A is called an *accessible automaton* if all its states are accessible. Thus, in an accessible automaton we can always find a path from the initial state to a given state.

Given a state machine A, all the non-accessible states can be removed without affecting the language accepted by A. The resulting machine is called the *accessible part* of A and will be denoted by $Acc(A)$.

Definition 7.1.4.
Let A be a state machine and let $V \subseteq \text{seq}(Input)$. Then we define an equivalence relation $\sim_V$ on Q by:

$q \sim_V q' \Leftrightarrow$

$\forall\ input^* \in V$, ($\exists\ input^*$, a path in A that starts in $q \Leftrightarrow \exists\ input^*$, a path in A that starts in q')

What this means is that for every path labelled by an element of V from q there is a path labelled by that element from q' and conversely.

If $q \sim_V q'$ then we say that q and q' are *V-equivalen*t. Otherwise we will say that *V distinguishes* between q and q'. If $V = \text{seq}(Input)$ then we say that q and q' are *equivalent*.

For two state machines $A = (Input, Q, \mathbf{F}, q_0)$ and $A' = (Input, Q', \mathbf{F}', q_0')$ over the same input alphabet, we say that A and A' are *equivalent* if their initial states q_0 and q_0' are equivalent. Since A and A' have only terminal states, A and A' are equivalent if and only if they accept the same language.

Definition 7.1.5.
A state machine A is called *reduced* if $\forall\ q, q' \in Q$ if q and q' are equivalent then $q = q'$. Given a state machine A, the machine constructed by merging the states of A that are equivalent will be called the *reduced machine* of A and will be denoted by $Red(A)$.

As shown previously, the merger of states in a state machine can be described in terms of a state machine morphism. Thus there exists a surjective state machine morphism from A to $Red(A)$.

Definition 7.1.6.
A deterministic state machine A is called *minimal* if it is accessible and reduced.

Theorem 7.1.1.
Given a state machine A, there is a minimal state machine that accepts the same language as A and this is unique up to a state machine morphism. We will call this the *minimal machine of A*, denoted $Min(A)$.

The minimal machine of an automaton A can be obtained by reducing $Acc(A)$ or by taking the accessible part of $Red(A)$ since the above result will ensure that the following diagram commutes (that is either way round gives the same result).

$$\begin{tikzcd}
A \arrow[r, "Red"] \arrow[d, "Acc"'] & Red(A) \arrow[d, "Acc"] \\
Acc(A) \arrow[r, "Red"] & Min(A)
\end{tikzcd}$$

Also, since $Min(A) = Red(Acc(A))$ there exists a surjective morphism from $Acc(A)$ to $Min(A)$ that merges the equivalent states of $Acc(A)$.

In Figure 7.1, $A_2 = Acc(A_1)$ and $A_3 = Red(A_2)$ thus $A_3 = Min(A_1)$. Also it is easy to see that g: $A_2 \rightarrow A_1$ defined by

$g(A_2) = A_1$, $g(B_2) = B_1$, $g(C_2) = C_1$, $g(D_2) = D_1$

is an injective state machine morphism and that h: $A_2 \rightarrow A_3$ defined by

$h(A_2) = A_3$, $h(B_2) = B_3$, $h(C_2) = C_3$, $h(D_2) = B_3$

is a surjective state machine morphism.

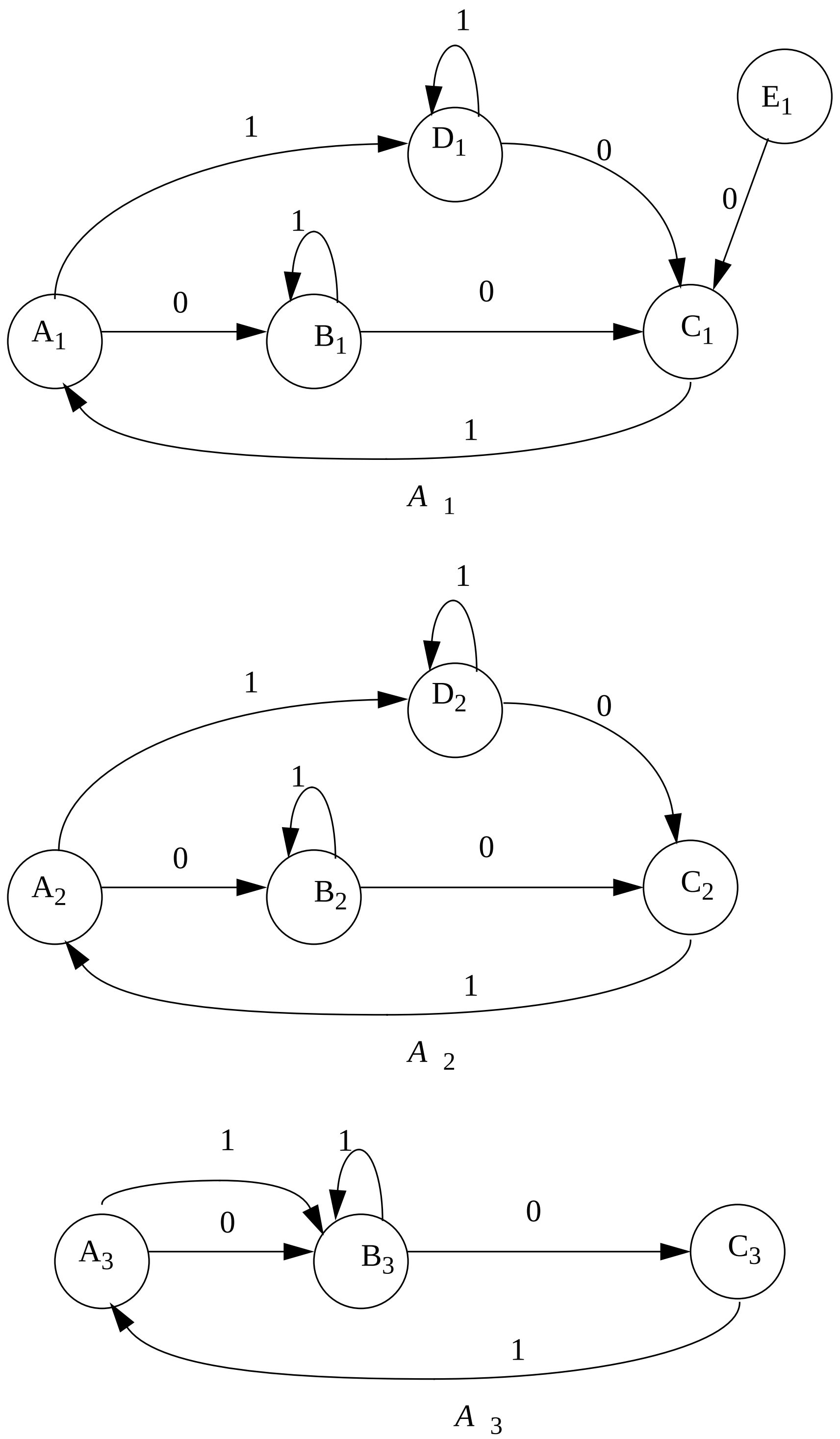

Figure 7.1 State machines and reduced machines.

State equivalence.
The following results will be used later in Chapter 8.

Lemma 7.1.2.
Let $A = (Input, Q, \mathbf{F}, q_0)$ and $A' = (Input, Q', \mathbf{F}', q_0')$ be two state machines *over the same input alphabet* and e: Q ↔ Q' a relation. Then A and A' are equivalent if the following are true.

1. $q_0\ e\ q_0'$
2. if $q\ e\ q'$ then $\forall\ input \in Input$, $(\exists\ input_1: q \rightarrow \theta$ an arc in $A) \Leftrightarrow$ $(\exists\ input_1: q' \rightarrow \theta'$ an arc in A'),

where θ e θ'.

Proof:
By induction the length of $input^* \in$ seq($Input$) it follows that
$\forall\ input^* \in$ seq($Input$) · $(\exists\ input^*: q_0 \rightarrow \theta$ a path in $A) \Leftrightarrow (\exists\ input^*: q_0' \rightarrow \theta'$ a path in A'),
where θ e θ'.

Definition 7.1.7.
Let $A = (Input, Q, \mathbf{F}, q_0)$ and $A' = (Input, Q', \mathbf{F}', q_0')$ be two state machines over the same input alphabet and let $q \in Q$ and $q' \in Q'$. Then q and q' are said to be *right-congruent* if there exists $input^* \in$ seq($Input$) so that $input^*: q_0 \rightarrow q$ is a path in A and $input^*: q_0' \rightarrow q'$ is a path in A'.

Lemma 7.1.3.
Let $A = (Input, Q, \mathbf{F}, q_0)$ and $A' = (Input, Q', \mathbf{F}', q_0')$ be two state machines over the same input alphabet and $e: Q \leftrightarrow Q'$ a relation defined by

$q\ e\ q' \Leftrightarrow q$ and q' are right-congruent.

If A and A' are equivalent then the following are true.

1. $q_0\ e\ q_0'$
2. if $q\ e\ q'$ then $\forall\ input \in Input \cdot (\exists\ input: q \rightarrow \theta$ an arc in $A) \Leftrightarrow$ $(\exists\ input: q' \rightarrow \theta'$ an arc in A'), where $\theta\ e\ \theta'$.

Proof:
1. Obvious.
2. Follows from the fact that q and q' are equivalent.

7.2. Theoretical basis of finite state machine testing.

The next section is devoted to finite state machine testing. Several methods of generating tests sets from a finite state machine specification exist. A general testing theory for finite state machines was developed by Chow, [76]. It assumes that the specification and the implementation can be both expressed as finite state machines and shows how a test set that finds all the faults of the implementation can be generated. There are a number of other approaches, most of them are quite restrictive, however. Some require that the specification and the implementation are finite state machines with the same number of states (see Sidhu et al. [93]); others assume that the specification is a finite state machine with special properties (see Bhattacharrya [94]).

In particular, testing strategies may attempt to identify the following types of faults:

missing states;
extra states;
missing transitions;
mis-directed transitions;
transitions with faulty functions (inputs/outputs);
extra transitions.

The *transition tour* method, [95], relies on the specification machine being minimal, strongly connected and complete. The method involves a traversal of all transitions without trying to target specific states. Efficient algorithms for determining minimal length sequences have been described, [96].

The *unique input-output (UIO) sequence* method, [96], involves deriving a sequence for each state which reflects the behaviour of that state. A number of improvements and variants of this method have been found. This method identifies that all the required states are present in the implementation.

The *W method* ,[76], is designed for the case where there may be more states in the implementation than in the specification. This is a potential advantage for this method over the others. However a number of variations and hybrid techniques are being developed. Some of these methods produce rather shorter sequences than the W method. This is an advantage if time for testing is short or more is known about the properties of the implementation (for example, it has the same number of states as the specification).

We will focus on the adaptation of the W method because of its capability of detecting more faults in an implementation than the others. Although we do give estimates, later, of the cost of the testing method we develop we do not focus on the question of the optimisation of the test sequences or the test set. Some of the work in [77] and [97] may generalise to the stream X-machine case.

The basis of Chow's test set generation are the concepts of characterisation set, state cover and transition cover of a minimal finite state machine. These will be defined next. Note that the minimality of the automaton will ensure their existence.

Definition 7.2.1.
Let $A = (Input, Q, \mathbf{F}, q_0)$ be a minimal finite state machine. Then $S \subseteq \mathrm{seq}(Input)$ is called a *state cover* set of A if $\forall\, q \in Q\ \exists\, input^* \in S$ so that $input^*: q_0 \rightarrow q$ is a path in A from the initial state q_0 to q.

So a state cover is a set of input sequences that enables us to access any state in the machine from the initial state.

Definition 7.2.2.
Let $A = (Input, Q, \mathbf{F}, q_0)$ be a minimal finite state machine. Then $T \subseteq \mathrm{seq}(Input)$ is called a *transition cover* of A if $\forall\, q \in Q\ \exists\, input^* \in T$ so that $input^*: q_0 \rightarrow q$ is a path in A from the initial state q_0 *to* q and $\forall\, input \in Input, input^* :: input \in T$.

In other words, for any state q there are sequences in T that take A to q from q_0 and then attempt to exercise all possible arcs from q irrespective of whether such arcs exist or not. It is easy to see that if S is a state cover of A then $T = S \cup [S \bullet Input\,]$ is a transition cover of A . Conversely, for any transition cover T there exists a state cover S with $S \cup [S \bullet Input\,] \subseteq T$.

Definition 7.2.3.

Let $\mathcal{A} = (Input, Q, \mathbf{F}, q_0)$ be a minimal finite state machine. Then $W \subseteq \text{seq}(Input)$ is called a *characterisation set* of $\mathcal{A}$ if W distinguishes between any two distinct states of $\mathcal{A}$.

Note that Chow's theory was developed in the context of finite state machines with outputs, i.e. an edge

$$q \xrightarrow{input/output} q'$$

is labelled by a pair *input/output* with $input \in Input$ and $output \in Output$; *output* is the output symbol and *Output* is called the output alphabet. In this case a path will be a sequence of input/output pairs and the definitions of state equivalence and distinguishability will refer to such input/output sequences rather than merely to sequences of inputs. However, Chow's theory remains valid for automata without outputs and this is the case we shall consider in our theoretical discussion.

For two automata $\mathcal{A}$ and $\mathcal{A}'$ over the same input alphabet, a set of input sequences will be called a *test set* of $\mathcal{A}$ and $\mathcal{A}'$ if its successful application to the two automata will ensure their equivalence.

Definition 7.2.4.
Let $\mathcal{A} = (Input, Q, \mathbf{F}, q_0)$ and $\mathcal{A}' = (Input, Q', \mathbf{F}', q_0')$ be two finite state machines over the input alphabet *Input*. Then a set $X \subseteq \text{seq}(Input)$ is called a *test set* of $\mathcal{A}$ and $\mathcal{A}'$ if the following is true:
if q_0 and q_0' are X-equivalent as states in $\mathcal{A}$ and $\mathcal{A}'$ respectively then $\mathcal{A}$ and $\mathcal{A}'$ are equivalent.

The idea of a test set is that it is a set of input sequences that can be used to establish whether two finite state machines are equivalent. If they are not equivalent, in other words if their behaviour is different, then we can find an input sequence in the test set that will show this difference in behaviour. The key objective, then is to find ways of constructing test sets. Clearly, seq(*Input*) is a test set but not a very useful one since it is infinite. We want to find *finite* test sets.

The following theorem is the basis of Chow's finite state machine testing, it describes a procedure for constructing a finite test set.

Theorem 7.2.1. ([76]).
Let $\mathcal{A}$ and $\mathcal{A}'$ be two minimal finite state machines over the input alphabet *Input*, n the number of states of $\mathcal{A}$ and n' the number of states of $\mathcal{A}'$. Let T and W, respectively, be a transition cover and a characterisation set of $\mathcal{A}$, $Z = Input^k \bullet W \cup Input^{k-1} \bullet W \cup \ldots \cup W$ and let $X = T \bullet Z$.
If $\text{Card}(Q') - \text{Card}(Q) \leq k$ and $\mathcal{A}$ and $\mathcal{A}'$ are X-equivalent, then $\mathcal{A}$ and $\mathcal{A}'$ are isomorphic.

The idea is that the transition cover T ensures that all the states and all the transitions of $\mathcal{A}$ are also present in $\mathcal{A}'$ and Z ensures that $\mathcal{A}'$ is in the same state as $\mathcal{A}$ after each transition is performed. Notice that Z contains W and also all sets $Input^i \bullet W$, i = 1, ..., k. This ensures that $\mathcal{A}'$ does not contain extra states. If there were up to k extra states, then each of them would be reached by some input sequence of up to length k from the existing states.

If the system specification and implementation can be *both* modelled as finite state machines $\mathcal{A}$ and $\mathcal{A}'$ then the set $X = T \bullet Z$ of the above theorem will ensure that these are equivalent provided that the maximum number of states of the implementation can be estimated. Note that the finite state machine model of the implementation, *A* ', need not be minimal since the above theorem can be applied to *A* and *Min*(*A* '). Hence *A* and *Min*(*A* ') are isomorphic, thus *A* and *A* ' are equivalent.

Corollary 7.2.1.
Let *A* and *A* ' be two finite state machines over the input alphabet *Input* with *A* minimal, Card(*Q*) the number of states of *A* and Card(*Q'*) the number of states of *A* '. Let *T, W, Z* and *X* be as in theorem 7.2.1. If Card(*Q'*) - Card(*Q*) $\leq k$ and *A* and *A* ' are *X*-equivalent, then *X* is a test set of *A* and *A* '.

The construction of the test set and its complexity.
Since the concepts of characterisation set and transition cover will be used later on in our testing theory, we will describe their construction in detail. In what follows we will refer to a finite state machine *A* with *n* states and *p* input symbols.

Constructing a transition cover
One way to construct a transition cover is to build a *testing tree*. *A* procedure of constructing testing trees is presented below.

1. Label the root of the tree with q_0, the initial state of *A* . This is the first level of the tree.
2. Suppose we have already built the tree up to a level *m*. Then the (*m* +1)'th level is built by examining nodes in the *m*'th level from left to right. A node at the *m*'th level is designated a terminal if its label is “Undefined” or is the same as a non-terminal at some level *l*, $l \leq m$. Otherwise let *q* denote its label. If on an input value *input*, the machine *A* goes from the state *q* to the state θ, we attach a branch and the successor node to the node labelled with *input* and θ, respectively. If there is no transition defined for *input* from *q* then we also attach a branch labelled *input*, but in this case the successor node will be labelled “Undefined”.

Obviously, the procedure above terminates since there are only a finite number of states in *A* . In fact, the tree has at most $n+1$ levels, where *n* is the number of states. Also, depending on the order in which we place the successor nodes, a different tree may result.

A transition cover *T* results by enumerating all the partial paths in the tree and adding the empty sequence to the set obtained in this way.

It is easy to see that *T* can be written as $T = S \cup [S \bullet Input]$, with *S* a state cover of *A* . The number of sequences of the resulting transition cover is $n.p +1$. It is also clear that this is the minimum possible number of elements of any transition cover.

A testing tree of *A* from Figure 7.2 may be found in Figure 7.3. Thus a transition cover will be:

T = {< >, 0, 1, 0::0, 0::1, 0::0::0, 0::0::1, 0::0::0::0, 0::0::0::1}

It is easy to see that

S = {< >, 0, 0::0, 0::0::0}

is a state cover of *A* and that $T = S \cup [S \bullet \{0, 1\}]$

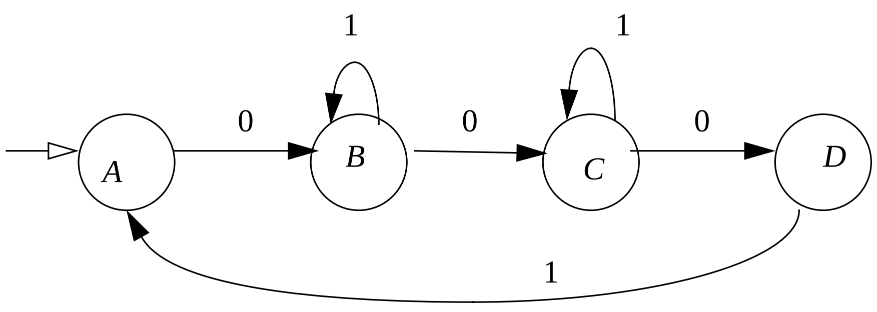

Figure 7.2 A simple finite state machine.

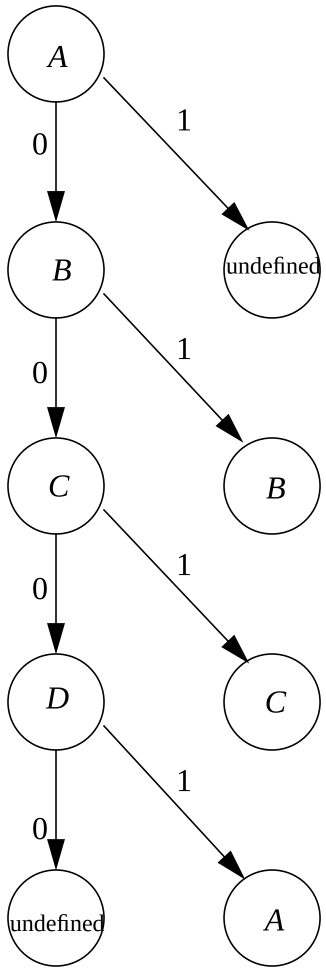

Figure 7.3 The testing tree for the machine in Figure 7.2.

Building a characterisation set.
There are many ways of constructing characterisation sets. We will present a procedure that gives the best possible result in the worst case scenario. First let us make some theoretical considerations.

Let $V, V' \subseteq$ seq(*Input*) be two sets of input sequences and $\sim_V$ and $\sim_{V'}$ respectively the equivalence relations on Q determined by them. Then we say that $\sim_V < \sim_{V'}$ if the following are true:

1. $\forall\ q, q' \in Q$ if $q \sim_{V'} q'$ then $q \sim_V q'$.
2. $\exists\ q, q' \in Q$ such that $q \sim_V q'$ and $\neg(q \sim_{V'} q')$.

On the other hand, if $\forall\ q, q' \in Q$, $q \sim_V q'$ if and only if $q \sim_{V'} q'$ then we say that $\sim_V = \sim_{V'}$. Also, if V is seq_i(*Input*), the set of all sequences of i inputs, then $\sim_V$ will be denoted by $\sim_i$. Then, for a minimal finite state machine A with n states, there exists $j \le n\text{-}1$ such that $\sim_1 < \sim_2 < \ldots\ \sim_j = \sim_{j+1} = \sim_{j+2} =$ ..... .

This is a well known result, a proof can be found in Eilenberg, [28]. Since A is minimal, seq_{n-1}(*Input*) will distinguish any pair of states of A . (Recall that seq_{n-1}(*Input*) is the set of all sequences of inputs from *Input* of length at most *n-1*.)

We can now give the following algorithm that finds a characterisation set W.

Step 1. Initialise $V = \emptyset$ and $i = 1$.
Step 2. (a) If $\sim_V < \sim_i$ then find $input^* \in \text{seq}_i$(*Input*) that distinguishes between two states q and q' that are not V-distinguishable; the partition determined by $\sim_i$ on Q and $input^*$ can be determined from the so-called P_k tables, see Gill [98]. Then V will become $V \cup \{input^*\}$ and step 2 is repeated.
(b) Otherwise, go to step 3.
Step 3. (a) If V does not distinguish between any pair of states of A , then increment i.
(b) Otherwise $W = V$ is the characterisation set required.

Using a simple induction it is easy to prove that the characterisation set W constructed by the algorithm will satisfy the following requirements.
1. Card(W) $\le n\text{-}1$;
2. $\forall\ i = 1, \ldots, n\text{-}1\ \exists$ at most $n\text{-}i$ elements of W of length at least i.

Hence, in the worst case the above algorithm will generate a characterisation set $W = \{input_1^*, \ldots, input_{n-1}^*\}$ with length($input_i$) = i, $i = 1, \ldots, n\text{-}1$.

On the other hand, for any n and p there exists a minimal finite state machine A with n states and p inputs so that any characterisation set of A will contain $n\text{-}1$ sequences $input_1^*, \ldots, input_{n-1}^*$ with $\forall\ i = 1 \ldots n\text{-}1$, length($input_i$) $\ge i$; for $p = 2$ such an automaton may be found in Figure 7.4. Thus the above algorithm will produce the best result in the worst case.

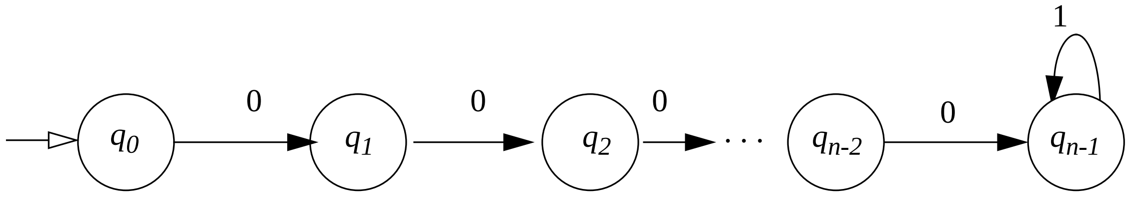

Figure 7.4 The worst case scenario.

The application of the algorithm on the automaton represented in Figure 7.4 will produce the characterisation set:

$W = \{1, 0::1, 0::0::1,..., 0^{n-2}::1\}$

Complexity results.
For a minimal machine with n states and p inputs the effort required in constructing T and W is roughly proportional to $n^2 \, . \, p$. This can be seen as follows: T is obtained by first constructing a testing tree and then enumerating the partial paths in the tree. Since for each state q and each input symbol *input* the transition from q on *input* appears exactly once in the transition tree, the complexity of the former is proportional to $n \, . \, p$. The complexity of the latter is also proportional to $n \, . \, p$ since there are $n \, . \, p + 1$ partial paths in the tree.

The transition set may be obtained from the so-called P_k tables using the above algorithm. The amount of work required to construct a P_k table is proportional to $n \, . \, p$, the number of entries in the table. There are at most $n - 1$ such tables, thus the effort required to construct them is proportional to $n^2 \, . \, p$.

Upper bounds for the test set size.
Using the above considerations regarding the sizes of a transition cover and of a characterisation set it can be deduced easily (see also Ipate [92]) that the number of test sequences required is less than $n^2 \, . \, p^{k+2} / (p-1) \approx n^2 \, . \, p^{k+1}$ and the total length of the test set, that is the size of the sequence obtained by putting all the test input sequences into one long sequence (a measure of the effort needed to fully test the system), is not greater than $n^2 \, . \, (n+k) \, . \, p^{k+2}/ (p-1) \approx n^2 \, . \, (n+k) \, . \, p^{k+1}$, where k is the difference between the number of states of the implementation and the number of states of the specification.

Various authors (see, for example, [96]) have devised improved methods based on the use of fewer input sequences in the test set or by combining test sequences together to improve efficiency. Usually, the gain in obtaining a smaller test set is partly offset by the increased complexity of the process of generating it.

7.3. Theoretical basis of stream X-machine testing.

This section will provide the theoretical basis for the stream X-machine testing method (SXMT). The method works on the assumption that the system specification and implementation can be both modelled as stream X-machines with the same type Φ and Φ satisfies some “design for test conditions” as presented in detail in what follows.

Thus let $M = (Input, Output, Q, Memory, \Phi, \mathbf{F}, q_0, m_0)$ be a stream X-machine with type Φ.

Design for test conditions.
The method reduces the task of testing that the two machines compute the same function to that of testing that their associated automata are equivalent. If this is to work then the basic processing functions Φ will have to meet two requirements, completeness w.r.t. *Memory* and output-distinguishability, these ideas are defined next.

Definition 7.3.1.

A type Φ is called *output-distinguishable* if the following is true.

$\forall \phi_1, \phi_2 \in \Phi \; \forall m \in \mathit{Memory}, \mathit{input} \in \mathit{Input}$ such that if $(m, \mathit{input}) \in$ domain ϕ_1, $(m, \mathit{input}) \in$ domain ϕ_2 and $\pi_1(\phi_1(m, \mathit{input})) = \pi_1(\phi_2(m, \mathit{input}))$ then $\phi_1 = \phi_2$.

What this is saying is that any two different basic processing functions will produce different outputs on some memory/input pair.

Definition 7.3.2.
A processing function $\phi \in \Phi$ is called *test-complete w.r.t. Memory,* (*t-complete*), if the following is true:

$\forall m \in \mathit{Memory} \; \exists \mathit{input} \in \mathit{Input}$ with $(m, \mathit{input}) \in$ domain ϕ.

A *type* Φ is called *test-complete w.r.t. Memory,*(*t-complete*), if $\forall \phi \in \Phi \; \phi$ is t-complete w.r.t. *Memory.*

In other words, any basic function will be able to process all memory values using a suitable input.

These two conditions are required of the specification machine and they will be referred to as “design for test conditions”. Although these might appear as being quite restrictive, they can be easily introduced into a specification by simply extending the definitions of the Φ functions in a suitable manner, introducing extra input symbols and augmenting the output alphabet ([92]). In very simple terms, this can be done as follows:

let $\phi \in \Phi$ be a processing function that is not complete w.r.t. *Memory* and let $\alpha \notin \mathit{Input}$ be an extra input (to be used specifically for testing process) and $g_0 \in \mathit{Output}$ an output chosen arbitrarily. Then the basic function ϕ_e defined by:

$$\phi_e(m, s) = \begin{cases} \phi(m, s), \text{ if } (m, s) \in \text{ domain } \phi \\ (g_0, m), \text{ if } s = \alpha \text{ and } m \in \mathit{Memory} \end{cases}$$

is complete w.r.t. *Memory.*

Obviously, ϕ_e will be a processing function of a stream X-machine whose input alphabet includes $\mathit{Input} \cup \{\alpha\}$ and has the same *Memory* as the original machine.

The above extension will be performed for *all* non-complete ϕs while the complete ϕs will remain unchanged; the resulting type will be named Φ_e.

If the resulting machine is to remain deterministic, then any two ϕs that are used as labels for arcs emerging from the same state will use different extra inputs. Thus the maximum number of extra inputs required will be at most:

$N_{in} = \text{Card}\{\phi' \in \Phi | \; \exists q \in Q$ such that ϕ and ϕ' are arcs emerging from $q\}$

In most cases $N_{in} << \text{Card}(\Phi)$ so the number of extra inputs is usually small.

The output-distinguishability will be achieved by a further augmentation process: the output alphabet of the resulting machine will become $Output \times G$, where the "*G*" component of the output will be used to distinguish between any two elements of Φ_e. Further details can be found in Ipate [92].

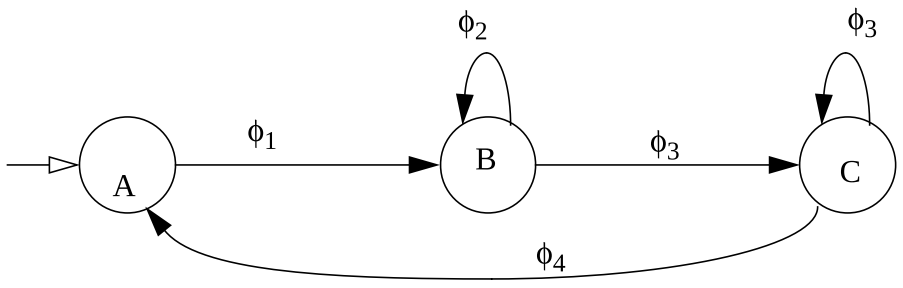

Figure 7.5 A simple stream X-machine.

Example 7.3.1.
We illustrate the above procedure with the following example. Let *M* be the stream X-machine represented in Figure 7.5 with *Input* = {x, y}, *Output* = {a, b}, *Memory* = {0, 1}, $m_0 = 0$ and the processing functions defined by:

$\phi_1(m, x) = (a, 0),\ m \in Memory$

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

$\phi_3(1, y) = (b, 1)$

$\phi_4(1, x) = (b, 0)$

If we augment *Input* to $Input_e = \{x, y, z, w\}$ then the extended type $\Phi_e = \{\phi_1, \phi_{2e}, \phi_{3e}, \phi_{4e}\}$ is complete w.r.t. *Memory,* where $\phi_{2e}, \phi_{3e}, \phi_{4e}$ are defined by:

$\phi_{2e}(0, x) = (a, 0)$
$\phi_{2e}(m, z) = (a, m),\ m \in Memory$

$\phi_{3e}(1, y) = (b, 1)$
$\phi_{3e}(m, w) = (a, m),\ m \in Memory$

$\phi_{4e}(1, x) = (b, 0)$
$\phi_{4e}(m, z) = (a, m),\ m \in Memory$

Furthermore, if the output alphabet is augmented to $\{a, b\} \times \{c, d, e\}$ then the augmented type $\Phi' = \{\phi_1', \phi_2', \phi_3', \phi_4'\}$ will be complete w.r.t. *Memory* and output-distinguishable.

$\phi_1'(m, x) = ((a, c), 0),\ m \in Memory$

$\phi_2'(0, x) = ((a, d), 0)$

$\phi_2'(m, \mathrm{z}) = ((\mathrm{a}, \mathrm{d}), m), m \in$ *Memory*

$\phi_3'(1, \mathrm{y}) = ((\mathrm{b}, \mathrm{d}), 1)$
$\phi_3'(m, \mathrm{w}) = ((\mathrm{a}, \mathrm{d}), m), m \in$ *Memory*

$\phi_4'(1, \mathrm{x}) = ((\mathrm{b}, \mathrm{e}), 0)$
$\phi_4'(m, \mathrm{z}) = ((\mathrm{a}, \mathrm{e}), m), m \in$ *Memory*

Fundamental test function.
The last theoretical concept introduced is that of the fundamental test function of a stream X-machine, defined as a means of converting sequences of ϕ's into sequences of inputs. This will be used to test paths of the machine using appropriate input sequences.

Definition. 7.3.3.
Let M = (*Input*, *Output*, Q, *Memory*, Φ, **F**, q_0, m_0) be a stream X-machine with Φ t-complete w.r.t. *Memory* and let $q \in Q$ and $m \in$ *Memory.* A function $t_{q,m}$: seq(Φ) $\rightarrow$ seq(*Input*) will be defined recursively as follows:

1. $\mathrm{t}_{q,m}(< >) = < >$

2. For $n \geq 0$, the recursion step that defines $t_{q,m}(\phi_1::...::\phi_n::\phi_{n+1})$ as a function of $t_{q,m}(\phi_1::...::\phi_n)$ depends on the following two cases:

i. if $\exists$ a path $p = \phi_1::...:: \phi_n$ in M starting from q, then
$t_{q,m}(\phi_1::...:: \phi_n::\phi_{n+1}) = t_{q,m}(\phi_1::...:: \phi_n) :: s_{n+1}$,
with s_{n+1} chosen such that $(m_n, s_{n+1}) \in$ domain ϕ_{n+1}
where $m_n = \pi_2(|p|(m, t_{q,m}(\phi_1::...:: \phi_n))$ is the final memory value computed by the machine along the path p on the input sequence $t_{q,m}(\phi_1::...:: \phi_n)$. Note that such s_{n+1} since Φ is t-complete w.r.t. *Memory* there exists such s_{n+1}.

ii. otherwise,
$t_{q,m}(\phi_1::...:: \phi_n::\phi_{n+1}) = t_{q,m}(\phi_1::...:: \phi_n)$

Then $t_{q,m}$ is called a *test function* of M w.r.t. (q, m).
If $q = q_0$ and $m = m_0$ then $t_{q,m}$ is denoted by t and is called a *fundamental test function* of M .
If $m = m_0$ then $t_{q,m}$ is denoted by t_q .

In other words if $p = \phi_1::...:: \phi_n$ is a path in M from q to some state q_n, then $t_{q,m}(\phi_1::...:: \phi_n)$ will be an input sequence which, when applied in state q and with memory contents m, will cause the computation of the machine to follow this path, i.e. $t_{q,m}(\phi_1::...:: \phi_n) = s_1::...::s_n$, where s_1 exercises ϕ_1, ..., s_n exercises ϕ_n.

If there is no arc labelled ϕ_{n+1} from q_n, then $t_{q,m}(\phi_1::...:: \phi_n::\phi_{n+1}) = t_{q,m}(\phi_1::...:: \phi_n):: s_{n+1}$, where s_{n+1} is an input that will to exercise an arc labelled ϕ_{n+1} emerging from q_n, thus making sure that such an arc does not exist.

Also, $\forall\ \phi_{n+2}, ..., \phi_{n+k} \in \Phi$, $t_{q,m}(\phi_1::...::\phi_n::\phi_{n+1}::...::\phi_{n+k}) = t_{q,m}(\phi_1::...:: \phi_n::\phi_{n+1})$

therefore no attempt will be made to exercise the subsequent ϕ's.

Note that a test function is not uniquely determined, many different possible test functions exist.

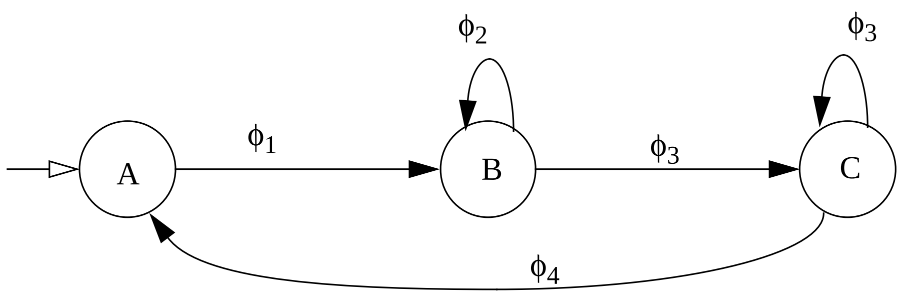

Figure 7.6 A stream X-machine.

Example 7.3.2.
Let *M* be the stream X-machine represented in Figure 7.6 with *Input* = {x, y}, *Output* = {a, b}, *Memory* = {0, 1}, m_0 = 0 and the processing functions defined by:

$\phi_1(m, \mathrm{y}) = (\mathrm{a}, 1)$, $m \in$ *Memory*

$\phi_2(m, \mathrm{x}) = (\mathrm{a}, 0)$, $m \in$ *Memory*

$\phi_3(m, \mathrm{y}) = (\mathrm{b}, 1)$, $m \in$ *Memory*

$\phi_4(m, \mathrm{x}) = (\mathrm{b}, 0)$, $m \in$ *Memory*

Then we can construct a fundamental test function with the following values:

$t\,(\phi_1) = \mathrm{y}$
$t\,(\phi_1::\phi_2) = \mathrm{y::x}$
$t\,(\phi_1::\phi_2::\phi_3) = \mathrm{y::x::y}$
$t\,(\phi_1::\phi_2::\phi_3::\phi_1) = \mathrm{y::x::y::y}$
$t\,(\phi_1::\phi_2::\phi_3::\phi_1::\phi_2) = \mathrm{y::x::y::y}$
$t\,(\phi_1::\phi_2::\phi_3::\phi_1::\phi_2::\phi_4) = \mathrm{y::x::y::y}$

The scope of a test function is to test whether a certain path exists or not in *M* using appropriate input symbols, hence the name. This idea is formalized in the following lemma.

Lemma 7.3.1.
Let *M* = (*Input*, *Output*, *Q*, *Memory*, Φ, **F**, q_0, m_0) and
M ' = (*Input*, *Output*, *Q*', *Memory*, Φ, **F**', q_0', m_0) be two stream X-machines with the same type Φ and initial memory, *A* and *A* ' their associated automata, *f* and *f '* the functions they compute and let t: seq(Φ) $\rightarrow$ seq(*Input*) be a fundamental test function of *M* and $X \subseteq \mathrm{seq}_+(\Phi)$. We assume that Φ is output-distinguishable and t-complete w.r.t *Memory.* If $\forall\ s^* \in t(X), f(s^*) = f'(s^*)$ then q_0 and q_0' are *X*-equivalent as states in *A* and *A* ' respectively.

Proof:
Let $\phi_1, ..., \phi_n \in \Phi$ with $f(t(\phi_1 ::...:: \phi_n)) = f'(t(\phi_1 ::...:: \phi_n))$. We prove that there exists a path $\phi_1 ::...:: \phi_n$ in A starting from q_0 if and only if there exists a path $\phi_1 ::...:: \phi_n$ in A' starting from q_0'. We have the following two cases.
1. There exists a path $\phi_1 ::...:: \phi_n$ in A starting from q_0 . Then by induction on $i = 1...n$ using the output-distinguishability of Φ it follows that there exists a path $\phi_1 ::...:: \phi_i$ in A' starting from q_0'.
2. There is no path $\phi_1 ::..:: \phi_n$ in A starting from q_0. Then let $i \geq 0$ the maximum number for which there exists a path $\phi_1 ::...:: \phi_i$ in A . Then there exists a path $\phi_1 ::...:: \phi_i$ in A' starting from q_0'. Let q_i and q_i', respectively, the end states of these paths. Now, if we assume that there exists an arc ϕ_{i+1} from q_i', from the output-distinguishability of Φ it will follow that there is an arc ϕ_{i+1} from q_i, which contradicts our initial assumption. Thus there is no path $\phi_1 ::...:: \phi_n$ in A' starting from q_0'.

We can now assemble our fundamental result which is the basis of the SXMT method.

Fundamental theorem of testing.
Theorem 7.3.1.
Let $M = (Input, Output, Q, Memory, \Phi, \mathbf{F}, q_0, m_0)$ and
$M' = (Input, Output, Q', Memory, \Phi, \mathbf{F}', q_0', m_0)$ be two stream X-machines with the same type Φ and initial memory, A and A' their associated automata, f and f' the functions they compute and let t: seq(Φ) $\rightarrow$ seq($Input$) be a fundamental test function of M . We assume that A and A' are minimal and that Φ is output-distinguishable and t-complete w.r.t *Memory.* Let also T and W, respectively, be a transition cover and a characterisation set of A , $Z = [\Phi^k \bullet W] \cup [\Phi^{k-1} \bullet W] \cup ... \cup W$, where k is a positive integer, $X = T \bullet Z$ and $Y = t(X)$. If Card(Q') - Card(Q) $\leq k$ and $\forall s^* \in Y$, $f(s^*) = f'(s^*)$ then A and A' are isomorphic.

Proof:
From Lemma 7.3.1 it follows that q_0 and q_0' are X-equivalent. The rest follows from theorem 7.2.1.

Notice that the completeness and output-distinguishability requirements can be relaxed to the attainable memory of M . That is, all memory values used by definitions 7.3.1 and 7.3.2 will be in *Mattain*(M).

Example 7.3.3.
For the machine of Example 7.3.2 we have
$T = \{< >, \phi_1, \phi_2, \phi_3, \phi_4, \phi_1::\phi_1, \phi_1::\phi_2, \phi_1::\phi_3, \phi_1::\phi_4, \phi_1::\phi_3::\phi_1, \phi_1::\phi_3::\phi_2, \phi_1::\phi_3::\phi_3, \phi_1::\phi_3::\phi_4\}$
$W = \{\phi_1, \phi_2\}$
$Z = [\{\phi_1, \phi_2, \phi_3, \phi_4\}^k \bullet \{\phi_1, \phi_2\}] \cup [\{\phi_1, \phi_2, \phi_3, \phi_4\}^{k-1} \bullet \{\phi_1, \phi_2\}] \cup ... \cup \{\phi_1, \phi_2\}$
If we take $k = 0$ then X = T $\bullet$ W thus
$X = \{\phi_1, \phi_1::\phi_1, \phi_2::\phi_1, \phi_3::\phi_1, \phi_4::\phi_1, \phi_1::\phi_1::\phi_1, \phi_1::\phi_2::\phi_1, \phi_1::\phi_3::\phi_1, \phi_1::\phi_4::\phi_1, \phi_1::\phi_3::\phi_1::\phi_1,$
$\phi_1::\phi_3::\phi_2::\phi_1, \phi_1::\phi_3::\phi_3::\phi_1, \phi_1::\phi_3::\phi_4::\phi_1, \phi_2, \phi_1::\phi_2, \phi_2::\phi_2, \phi_3::\phi_2, \phi_4::\phi_2, \phi_1::\phi_1::\phi_2,$
$\phi_1::\phi_2::\phi_2, \phi_1::\phi_3::\phi_2, \phi_1::\phi_4::\phi_2, \phi_1::\phi_3::\phi_1::\phi_2, \phi_1::\phi_3::\phi_2::\phi_2, \phi_1::\phi_3::\phi_3::\phi_2, \phi_1::\phi_3::\phi_4::\phi_2\}$
and
Y = {y, y::y, x, y, x, y::y, y::x::y, y::y::y, y::x, y::y::y, y::y::x, y::y::y::y, y::y::x::y, x, y::x, x, y, x, y::y,
y::x::x, y::y::x, y::x, y::y::y, y::y::x, y::y::y::x, y::y::x::x}
thus
Y = {y, y::y, x, y::x::y, y::y::y, y::x, y::y::x, y::y::y::y, y::y::x::y, y::x::x, y::y::y::x, y::y::x::x}

Note that we can simplify further by removing the sequences y::y::y, y::y and y since they all occur as an initial segment (prefix) of the sequence y::y::y::y.

If our aim is to ensure that the two machines compute the same function, then the minimality of A' is not really necessary. This result will be given next. Also, in a similar way to the case for a finite state machine, we will define a test set of two stream X-machines as a set of input sequences of which successful application to the two machines will ensure the equivalence of their associated automata.

Definition 7.3.4.
Let $M = (Input, Output, Q, Memory, \Phi, \mathbf{F}, q_0, m_0)$ and
$M' = (Input, Output, Q', Memory, \Phi, \mathbf{F}', q_0', m_0)$ be two stream X-machines, A and A' their associated automata and f and f' the functions they compute. Then $Y \subseteq \text{seq}(Input)$ is called a *test set* of M and M' if the following is true:

if $\forall s^* \in Y, f(s^*) = f'(s^*)$ then A and A' are equivalent.

Corollary 7.3.1.
Let $M = (Input, Output, Q, Memory, \Phi, \mathbf{F}, q_0, m_0)$ and
$M' = (Input, Output, Q', Memory, \Phi, \mathbf{F}', q_0', m_0)$ be two stream X-machines, A and A' their associated automata, f and f' the functions they compute and let t: seq(Φ) $\rightarrow$ seq($Input$) be a fundamental test function of M. We assume that A is minimal and that Φ is output-distinguishable and t-complete w.r.t. *Memory.* Let also *T, W, Z, X* and *Y* as in Theorem 7.3.1. If Card(Q') - Card(Q) $\leq k$ then X is a test set of M and M'.

Obviously, if A and A' are equivalent then $f = f'$. Thus if the application of Y to M and M' will produce identical results then the two machines are guaranteed to have the same input/output behaviour. Therefore, if the specification and the implementation can be both expressed as machines with the same type and the required design for testing conditions are met, then Y will be a test set that detects *all* faults of the implementation.

Note that since T can be written as $T = S \cup [S \bullet \Phi]$, with S a state cover of A then the test set Y can be also expressed as:

$$Y = t\,(S \bullet (\Phi^{k+1} \cup \Phi^k \cup ... \cup \Phi) \bullet W)$$

Expected outputs.
For most systems, M will compute a total function, i.e. for any sequence of inputs $s^* \in$ seq($Input$), $f(s^*)$ will be a sequence of outputs. In this case, the process of comparing the two output sequences, $f(s^*)$ and $f'(s^*)$, is straightforward. However, the method does not rely on f being a total function as long as it is clear what we mean by "$f(s^*)$ is undefined", i.e. what the system is supposed to do when it receives an input sequence s^* for which $f(s^*)$ is undefined. For example this could mean that the sequence s^* will cause the program specified by M to exit.

Complexity and upper bound of the test set.
The final question that needs to be addressed is concerned with the practicality of the method. For example, how complex is the test generation algorithm? It is clear that the complexity of the algorithm depends on the complexity of the basic functions in Φ. If the complexity of each ϕ is at most C, then the complexity of the algorithm that generates the test set $Y = t\,(X)$ will be proportional to the

product of *C*, the number of input symbols and the total size of *X*. Thus, the complexity will be proportional to $C \,.\, p \,.\, r^{k+1} \,.\, n^2 \,.\, (n+2k)$, where n = Card(Q), k = Card(Q') - Card(Q), p = Card(*Input*) and r = card(Φ).

The maximum number of test sequences required is less then $n^2 \,.\, r^{k+2} / (r - 1) \approx n^2 \,.\, r^{k+1}$ and the total length of the test set is less then $n^2 \,.\, (n+k) \,.\, r^{k+2} / (r - 1) \approx n^2 \,.\, (n+k) \,.\, r^{k+1}$. Note that these figures are the upper bounds, the actual number of inputs required may be much lower.

Test generation tools.

The process of constructing the complete functional test set for a stream X-machine is a lengthy and tedious process in all but the most trivial cases. It is thus important that the process should be automated. Bogdanov [99] has investigated this issue in some depth and has constructed a powerful test generation environment involving a number of tools that, together, will achieve the goal of automatic test set generation based on this theory. In this work the X-machine description can be extracted from a number of sources, one for example is based on the use of Statecharts [31] and STATEMATE [32] for the state diagram and the formal language Z for the transition semantics, see [33].