Chapter 4.

Testing.

This chapter consists of two main parts. The first reviews the main existing testing methods and discusses to what extent they achieve the ultimate goal of testing (i.e. to find all faults). The second part presents our testing method based on the stream X-machine model.

4.1. Background.

The two major objects used in System development activities are *specification* and *implementation*. Most of these activities concern the conversion of the specification into an implementation. But others are concerned with evaluating how well the implementation satisfies the specification. If the specification S and the implementation I are assumed to be (partial) functions S, I: $D \rightarrow R$, then we say that the implementation is *correct* w.r.t. the specification if $S(x) = I(x), \forall x \in D$. Conversely, a *failure* occurs in the implementation if, for an input x, the output produced by the implementation does not correspond to that produced by the specification. Any part of the implementation that could lead to a failure is a *fault*. Then, the implementation is correct w.r.t. the specification iff it is fault-free.

Testing attempts to achieve correctness by detecting *all* the faults that are present in the implementation so they can be removed. A *finite* set of inputs $X \subseteq D$ is designed and the result produced by each element of I (i.e. I(x)) is compared with the expected result (i.e. S(x)). The set of inputs X will be called the *test set*. The elements of the test set are chosen subject to a particular *criterion*. Many techniques for carrying out testing, and in particular for the generation of test sets exist and some of them are supported by automatic tools. There are also many ways of classifying these techniques according to the particular criterion used. The most common classification is into *program based* techniques and *functional* techniques. There are also random methods which generate test sets randomly and some statistical methods that combine random generation with one of the other techniques (e.g. Waeselynck [57]) Analysis methods have also been developed that estimate the probability of an implementation being correct after the testing has been successfully completed. There are a number of different types of statistical models used (Miller et al. [43], Hamlet & Taylor [23], Weiss & Weyuker [58]) and they lead to conflicting claims as to the benefits of different types of testing.

4.1.1. Program based testing

These techniques are also known as *structural* and *white-box* testing. Program based testing methods base their test selection criterion on the structure of the finished code. There is a well defined hierarchy of criteria (see Myers [47] or Ntafos [49]) the most common ones being described here in ascending order of strength.

Statement (or segment) coverage: If the test causes every statement of the code to be executed at least once, then statement coverage is achieved.

A segment is an indivisible piece of code, no part can be executed without all of it being executed.

Branch coverage: If the test causes every branch to be executed at least once, then branch coverage is achieved. In other words, for every branch statement, each of the possibilities must be performed on at least one occasion.

Path coverage: If the test set causes every distinct execution path to be taken at some point, then path coverage is achieved. E.g., in the case of a loop, there are paths for each number of iterations of the loop. Even for quite short and simple programs, this level of coverage can be infeasible.

Limitations of program based testing

None of these program based methods use the requirements of the system in their test selection criterion. Instead they all make the assumption that the implementation matches the requirements in its broad structure. This can be a severe limitation if you consider that the ultimate goal of testing is to compare the implementation with its requirements.

Errors corresponding to missing paths in the code will not generally be detected.

Another drawback is that you have to wait until there is some of the actual code before you can begin to construct tests. This corresponds to a software life-cycle in which testing it carried out after the design and the implementation of the system has been completed. This is very expensive as it only uncovers faults long after they are introduced, especially faults in requirements. Testing requirements is an important issue and one in which a testing method based on a formal specification could be valuable.

Nevertheless, program based testing methods are still in widespread use (see Gelperin & Hetzel [17] or a testing standard such as [2]) and undoubtedly reveal a great many errors that might otherwise escape. Also, several tools that support these techniques exist (see CAST Report [4]).

4.1.2. Functional testing.

These methods are known as *black-box* testing .

Functional testing methods base their test case selection criteria largely on the intended functionality of the implementation, i.e. on the specification or requirements. This fits in well with the goal of comparing implementations with their requirements.

4.1.2.1. The category-partition method.

The most widely used functional technique is the *category-partition* method. The method was presented by Ostrand & Balcer, [51], and it involves several steps, the main ones being described below.

A. Analysis of the specification.

First, functional units that can be individually tested are identified; either top level user commands or functions that are called by them, or lower level functions are defined. Several stages of decomposition may be required.

For all the functions identified, find the parameters (i.e. explicit inputs or outputs to the functional unit, either by the program or by the user) that affect the function's behaviour.

Example. Consider the specification of a sorting program. The program is to accept an array of integers of variable length and is to display a permutation of the array, but with the values correctly ordered with respect to the normal order on integers. Then, the only parameters are the unsorted array as input and the sorted array as output.

B. Categorise the parameters.

For each parameter, identify properties and characteristics that have particular effects on the function's behaviour. Classify the characteristics of the parameters into categories that characterise the behaviour of the function.

Example. For the sorting program, the categories are the array's size and the order of the unsorted array.

C. Partition the categories into choices.

Determine the different significant cases that can occur within each parameter condition category. These cases are choices. Each choice consists of a subset of the category's values, which will all lead to the same behaviour. The choices must be mutually exclusive.

Example. Using the "array size" partition from above, the choices might be 0, 1, or > 1.

D. Generate the test set.

A test set is generated such that each choice of each category is satisfied by at least one input. This can be done automatically (see Ostrand & Balcer [51]).

There are some clear advantages of this method. Firstly, unlike the program based techniques, the test cases are derived from the specification. Hence, there is a better chance of detecting if some functionality is missing from the implementation (e.g. a missing path in a program). Also, the test phase can be started early in the development process and the test set can be easily modified as the system evolves.

The process of working through all of the details of the method may well reveal limitations of the design specification. If so, these should of course be addressed and then the implications considered for the parts of the testing process already carried out.

Limitations of the category-partition method.

It is difficult to describe formally the concepts of category and choice, so it is hard to assess how adequate the criteria used for choosing these are. As a result, the method relies heavily on the experience of the tester.

The method does not offer any guidance on combining the tests of individual functions into higher level tests that ensure that these functional units are correctly integrated.

Conclusions.

Several other functional testing methods exist (see Goodenough & Gerhart [21], Gerrard et al. [18], Myers [47, 48] or Hayes [25]), many of them being broadly similar to the category-partition method. All of them (and indeed the program based ones) share the same drawback in the sense that they do not enable us to make any statement about the number and the type of faults that remain undetected *after* testing is completed. In practice all we can usually say is that we have uncovered a number of faults over a period of testing effort and the graph of the number of faults against the period or amount of testing, measured suitably, indicates that the growth rate is reducing.

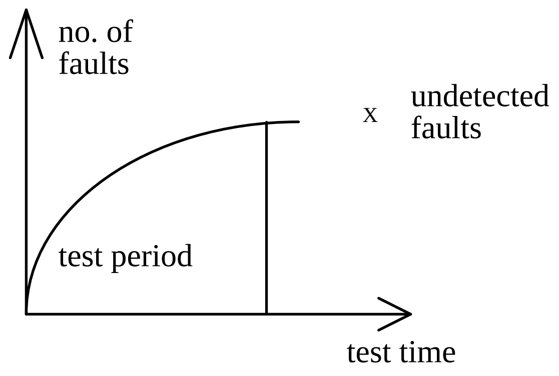


Figure 4.1.

The trouble is that we do not know that no further faults exist in the system at a particular time. A general formula for this curve is not known; if one existed for a certain testing method it would probably depend on the type of the system and on those doing and managing the testing as well as wider issues relating to the management of the design project, the implementation vehicle, the design methods and so on. It is therefore fair to say that making sure that systems are fault free is quite beyond current testing methods. "All they can tell is that a system has failed. They cannot tell us that the system is correct" (statement attributed to Dijkstra).

4.1.3. Theoretical testing.

The testing techniques discussed so far (and indeed most techniques used in practice) have been based on the experience of software developers rather than on a well founded theory.

Very few attempts have been made to address the issues of testing from a theoretical point of view. An outline theory of testing is introduced by Goodenough & Gerhart, [21], and developed further by Weyuker and Ostrand, [59]. They treat a software system as a (partial) function from an input set to an output set and the testing process consists of constructing "revealing sub-domains" that expose faults in this function (i.e. a sub-domain B of the function is revealing for a fault F if, whenever F affects any element of B it affects all the elements of B). This is a very general and abstract approach that provides an useful set of simple concepts and terminology for the discussion of some aspects of testing without giving too many clues of how to construct an effective testing strategy.

The paucity of attempts to address the theoretical issues of testing is not surprising. Indeed, recall that the (idealised) goal of testing is to find *all* the faults in the implementation. Let $S: D \to R$ and $I: D \to R$ the specification and the implementation of a system respectively and $X \subseteq D$ the (finite) test set. Then X finds all the faults if and only if $S(x) = I(x), \forall x \in X \Rightarrow S(x) = I(x), \forall x \in D$. In this case we say that the test set X is *adequate*. Then, it is clear that if S and I are partial functions computed by arbitrary computer systems (i.e. Turing machines), such a test set does not exist (i.e. if it did, the halting problem for Turing machines would be solvable). Hence, in this case the goal of testing is not attainable.

How can we get around this problem? A solution would be to develop testing methods based on more restrictive computational methods (e.g. finite state machines).

4.1.4. Finite state machine testing.

Some finite state machine testing methods exist. Most of them are quite restrictive; some require that the specification and the implementation are finite state machines with the same number of states (see Sidhu et al. [54]); others assume that the specification is a finite state machine with special properties (see Bhattacharrya [3]).

A more general testing theory for finite state machines was developed by Chow, [6]. It assumes that the specification and the implementation can both be expressed as finite state machines and shows how a test set that finds all the faults in the implementation can be generated.

4.1.4.1. Preliminary concepts.

Before we describe the method in more detail we introduce some concepts that we shall be needing later. The following definitions are largely from Chow, [6].

Definition 4.1.4.1.1.
Let $\mathcal{A} = (\Sigma, \Gamma, Q, F, G, q_0)$ be a finite state machine, $S \subseteq \Sigma^*$ a set of input sequences and $q, q' \in Q$ two states. Then we say that S *distinguishes* between q and q' if $\exists s \in S$ such that $G_e(q, s) \neq G_e(q', s)$.

In other words, s produces different outputs when applied to q and q' respectively.

Definition 4.1.4.1.2.
Let $\mathcal{A} = (\Sigma, \Gamma, Q, F, G, q_0)$ be a minimal finite state machine. Then a set of input sequences $W \subseteq \Sigma^*$ is called a *characterisation set* of $\mathcal{A}$ if W can distinguish between any two pairs of states of $\mathcal{A}$.

Definition 4.1.4.1.3.
Let $\mathcal{A} = (\Sigma, \Gamma, Q, F, G, q_0)$ be a minimal finite state machine. Then a set of input sequences $S \subseteq \Sigma^*$ is called a *state cover* if $\forall q \in Q, \exists s \in S$ such that $q = F_e(q_0, s)$ (i.e. s forces the machine $\mathcal{A}$ into q from the initial state q_0).

Definition 4.1.4.1.4.

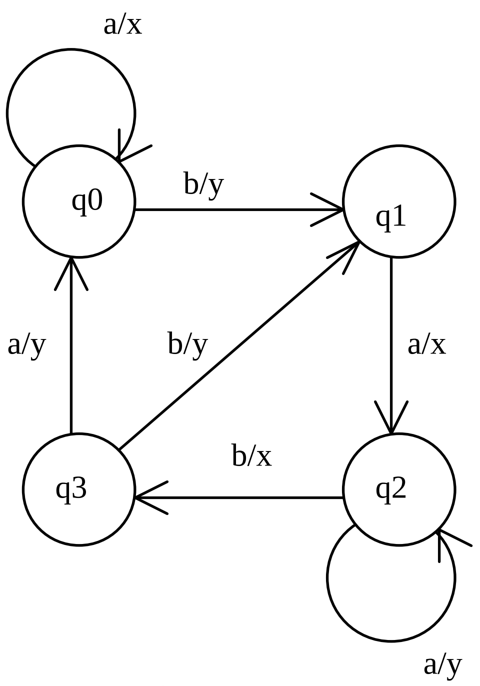
Let $\mathcal{A} = (\Sigma, \Gamma, Q, F, G, q_0)$ a minimal finite state machine. Then a set of input sequences $T \subseteq \Sigma^*$ is called a *transition cover* if $\forall\ q \in Q$, $\exists\ t \in \Sigma^*$ with $q = F_e(q_0, t)$ such that $t \in T$ and $t\sigma \in T$, $\forall\ \sigma \in \Sigma$ (i.e. t forces the machine into q from q_0 and $t \in T$ and $t\sigma \in T$, $\forall\ \sigma \in \Sigma$).

Notice that, since $\mathcal{A}$ is minimal, a characterisation set, a state cover and a transition cover of $\mathcal{A}$ exist.

It is clear that if S is a state cover of $\mathcal{A}$, then $T = S \cup S\Phi$ is a transition cover of $\mathcal{A}$. Also, for any transition cover T of $\mathcal{A}$, there exists a state cover S such that $S \cup S\Phi \subseteq T$.

Example 4.1.4.1.5.

Let $\mathcal{A}$ be the (minimal) finite state machine with $\Sigma = \{a, b\}$ and $\Gamma = \{x, y\}$ represented in figure 4.2.



Note: $q_i \xrightarrow{a/x} q_j$ denotes that $F(q_i, a) = q_j$, and $G(q_i, a) = x$.
Figure 4.2.

Then:

$W = \{a, b\}$ is a characterisation set of $\mathcal{A}$;
$S = \{1, b, ba, bab\}$ is a state cover of $\mathcal{A}$;
$T = \{1, a, b, ba, bb, baa, bab, baba, babb\}$ is a transition cover of $\mathcal{A}$.

The following theorem is from Chow, [6], and it represents the theoretical basis of his testing method.

Theorem 4.1.4.1.6.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, F, G, q_0)$ and $\mathcal{M}' = (\Sigma, \Gamma, Q', F', G', q_0')$ be two minimal finite state machines. Let T and W, respectively, be a transition cover and a characterisation set of $\mathcal{M}$ and $Z = \Sigma^k W \cup \Sigma^{k-1} W \cup ... \cup W$. If $card(Q') - card(Q) \leq k$ and q_0 and q_0' are TZ-equivalent, then $\mathcal{M}$ and $\mathcal{M}'$ are isomorphic.

The idea is that the transition cover T ensures that all the states and all the transitions of $\mathcal{M}$ are also present in $\mathcal{M}'$ and Z ensures that $\mathcal{M}'$ is in the same state as $\mathcal{M}$ after each transition is used. Notice that Z contains W and also all sets $\Sigma^i W$, i = 1, ..., *k*. This ensures that $\mathcal{M}'$ 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.

4.1.4.2. The state machine testing method.

The method relies on the following assumptions.

1. The specification is a minimal finite state machine $\mathcal{M}$.
2. The implementation can be modelled as a finite state machine $\mathcal{M}'$ with the same input and output alphabets as $\mathcal{M}$.
3. The number of states in $\mathcal{M}'$ is bounded by a certain number *n'*.

Under these circumstances X = TZ is a test set that finds *all* faults, where T and W are a transition cover and a characterisation set of $\mathcal{M}$ respectively,

$$Z = \Sigma^k W \cup \Sigma^{k-1} W \cup ... \cup W$$

and $k = n' - n$, where *n* is the number of states of $\mathcal{M}$ and *n'* is the (estimated) upper bound of the number of state of $\mathcal{M}'$.

Notice that $\mathcal{M}'$ need not be minimal. Indeed if $\mathcal{M}'$ is not minimal then we can apply theorem 4.1.4.1.6 for the minimal machine of $\mathcal{M}'$, Min($\mathcal{M}'$). Then $\mathcal{M}$ and Min($\mathcal{M}'$) will be isomorphic, hence $\mathcal{M}$ and $\mathcal{M}'$ will compute the same function.

4.1.4.3. Construction of the test set and complexity.

Since the concepts of characterisation set and transition cover will be used later on in our testing theory, we shall describe their construction in detail. In what follows we shall be referring to a finite state machine $\mathcal{M}$ with *n* states and *p* input symbols.

4.1.4.3.1. Transition cover.

One way to construct a transition cover is by building a testing tree. The procedure for constructing a testing tree given in what follows is largely from Chow, [6].

1) Label the root of the tree with the initial state of $\mathcal{A}$, q_o. 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) level is built by examining nodes in the *m*'th level from left to right. A node at the *m*'th level is terminal if its label is "Undefined" or is the same as a nonterminal at some level ℓ, $\ell \leq m$. Otherwise let q_i denote its label. If on an input σ, the machine $\mathcal{A}$ goes from the state q_i to the state q_j, we attach a branch and the successor node to the node labelled with σ and q_j, respectively. Otherwise (i.e. if there is no transition defined for σ from q_i), then we also attach a branch labelled σ, but in this case the successor node will be labelled "Undefined".

For the finite state machine from example 4.1.4.1.5, a testing tree is represented in figure 4.3.

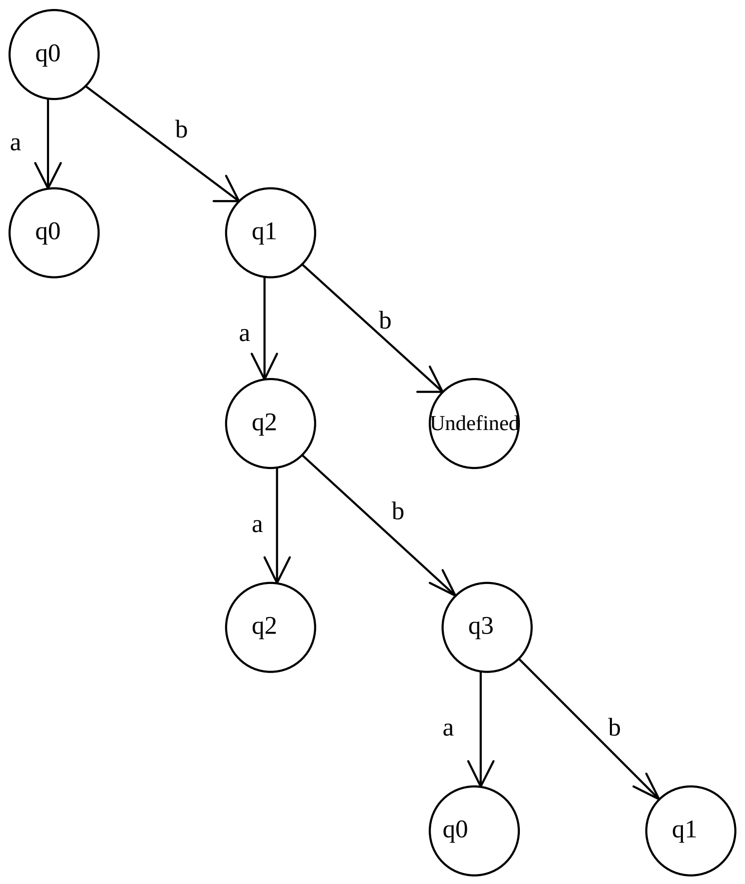


Figure 4.3.

Obviously, the procedure above terminates since there are only a finite number of states in $\mathcal{A}$. In fact, the tree has at most *n*+1 levels. Also, depending on the order in which we place the successor nodes, a different tree may result. A transition cover results by enumerating all the partial paths in the tree and adding the empty sequence to the set obtained in this way. The number of sequences of the resulting transition cover is $n \cdot p+1$, where p = card(Σ). It is also clear that this is the minimum possible number of elements of any transition cover.

4.1.4.3.2. Characterisation set.

There are many ways of constructing characterisation sets. We shall describe a procedure that gives the best possible result in the worst case scenario. This issue is not analysed in detail in Chow, [6]. First, let us make the following remark.

Let V, V' $\subseteq \Sigma^*$ 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:

i) $\forall$ q, q' $\in$ Q if $q \sim_{V'} q'$ then $q \sim_V q'$.

ii) $\exists$ q, q' $\in$ Q such that $q \sim_V q'$ and $\neg(q \sim_{V'} q')$.

If $\forall$ q, q' $\in$ Q, $q \sim_{V'} q'$ iff $q \sim_{V'} q'$, we say that $\sim_V = \sim_{V'}$.

Also, if $V = \bigcup_{j=1}^{i} \Sigma^j$, i $\in$ N, then $\sim_V$ will be denoted by $\sim_i$.

Then, for a minimal finite state machine $\mathcal{A}$ with n states, there exists j ≤ n-1 such that $\sim_1 < \sim_2 < \ldots \sim_j = \sim_{j+1} = \sim_{j+2} = \ldots\ldots$. This is a well known result, a proof can be found in Eilenberg, [12]. Since $\mathcal{A}$ is minimal, it follows that Σ^j will distinguish any pair of states in $\mathcal{A}$.

Let us now give the following algorithm that finds a characterisation set W.

Algorithm 4.1.4.3.2.1.

Step1. Initialise V = ∅ and i = 1.

Step2. (a) If $\sim_V < \sim_i$ then find $s \in \Sigma^i$ such that s distinguishes between two states q and q' that are not V-distinguishable (i.e. the partition determined by $\sim_i$ on Q can be determined using the so called P_k *tables* (see Gill [19]); also, s can be determined from these tables (see Gill [19], algorithm 4.1)). Then V will become V $\cup$ {s} and step2 is repeated.

(b). Otherwise, go to step3.

Step3. (a) If V does not distinguish between any pair of states of $\mathcal{A}$, then increment i.

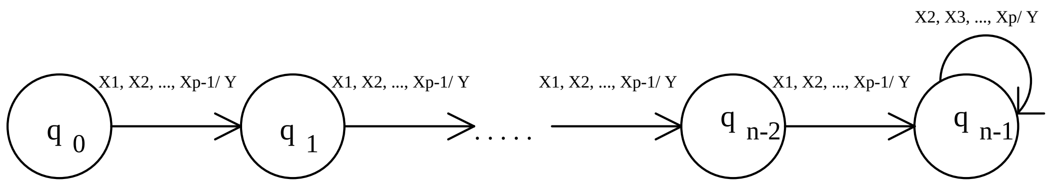
(b) Otherwise W = V is the characterisation set required.

Using a simple induction and the above remark it is easy to prove that the characterisation set W constructed by the algorithm will satisfy:

i) card(W) ≤ n-1;

ii) $\forall$ i $\in$ {1, ..., n-1}, $\exists$ at most n-i elements of W of length at least i.

Hence, it follows that in the worst case the above algorithm will generate a characterisation set W = $\{s_1,..., s_{n-1}\}$ such that $|s_i|$ = i, i = 1, .., n-1. This is the best result in the worst case scenario, since for any n and p there exists a minimal finite state machine $\mathcal{A}$ (see figure 4.4) with n states and p inputs such that if W' is a characterisation set of $\mathcal{A}$, then there exists W'' $\subseteq$ W', W'' = $\{s_1,..., s_{n-1}\}$, such that $|s_i| \geq$ i, i = 1, .., n-1.



$x_1, x_2, ..., x_{p-1}/ y$ denotes that the machine produces the output y on each of the inputs $x_1, x_2, ..., x_{p-1}$.

Figure 4.4.

4.1.4.3.3. Complexity.

For a minimal machine with n states and p inputs the effort required in constructing T and W is roughly proportional to $n^2 \cdot p$. 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 σ the transition from q on σ appears exactly once in the transition tree, the complexity of the former is proportional to $n \cdot p$. The complexity of the latter is also proportional to $n \cdot p$ since there are $n \cdot p + 1$ partial paths in the tree (see Chow [6]).

The transition set is obtained by first constructing the P_k and then applying algorithm 4.1.4.3.2.1. The amount of work required to construct a P_k table is proportional to $n \cdot p$, the number of entries in the table. Since there are at most n -1 such tables, the effort required to construct them is proportional to $n^2 \cdot p$. The input sequences required by algorithm 4.1.4.3.2.1 step2 (a) will be obtained using [19, algorithm 4.1]. The amount of work required to construct a sequence s using this algorithm will be proportional to $p \cdot |s|$ (i.e. each symbol in s is obtained by comparing the values of two columns in a P_k table (each column has p elements)). Since the total length of W is no more then $\frac{n(n-1)}{2}$, the total amount of work required to construct W is proportional to $n^2 \cdot p$.

4.1.4.4. Upper bounds for the test set size.

Since

$$\mathrm{card}(X) = \mathrm{card}(T) \cdot \mathrm{card}(\Sigma^k \cup \Sigma^k \cup \ldots \cup \{1\}) \cdot \mathrm{card}(W),$$

the maximum number of test sequences required will be:

$$\max(\mathrm{card}(X)) = (n \cdot p + 1)\ (1 + p + \ldots + p^k)(n - 1) = (n \cdot p + 1)\ \frac{p^{k+1} - 1}{p - 1}(n - 1).$$

Hence

$$\text{card}(X) \le n^2 \cdot \frac{p^{k+2}}{p-1}.$$

For large *p*,

$$\max(\text{card}(X)) \approx n^2 \cdot p^{k+1}.$$

An upper bound for the total length of the set test, $|X| = \sum_{s \in X} |s|$, can be determined by observing that

$$|X| = N_1 + N_2 + N_3,$$

where

$$N_1 = \text{card}(T) \cdot \text{card}(\Sigma^k \cup \Sigma^{k-1} \cup ... \cup \{1\}) \cdot |W|,$$
$$N_2 = \text{card}(T) \cdot |(\Sigma^k \cup \Sigma^{k-1} \cup ... \cup \{1\})| \cdot \text{card}(W),$$
$$N_3 = |T| \cdot \text{card}(\Sigma^k \cup \Sigma^{k-1} \cup ... \cup \{1\}) \cdot \text{card}(W).$$

Since

$$|T| \le p + 2 \cdot p + \ldots + n \cdot p = \frac{p \cdot n(n+1)}{2},$$
$$|(\Sigma^k \cup \Sigma^{k-1} \cup ... \cup \{1\})| = p + 2 \cdot p^2 \ldots + k \cdot p^k \le \frac{k \cdot p^{k+1}}{p-1},$$
$$|W| \le \frac{n(n-1)}{2},$$

we have

$$|X| \le (p \cdot n + 1)\frac{p^{k+1}}{p-1} \cdot \frac{n(n-1)}{2} + (p \cdot n + 1)\frac{k \cdot p^{k+1}}{p-1}(n-1) + \frac{p \cdot n \cdot (n+1)}{2} \cdot \frac{p^{k+1}}{p-1}(n-1)$$

Hence

$$|X| \le \frac{p^{k+2}}{p-1} \cdot n^2 \cdot (n+k) = \frac{p^{k+2}}{p-1} \cdot n^2 \cdot n'.$$

For large *p*, the upper bound for the total length of the test set is approximately

$$n' \cdot n^2 \cdot p^{k+1}.$$

4.1.4.5. Improvement in the test set size.

Fujiwara et al., [16], prove that the test set can be reduced to $X' = X_1 \cup X_2$, where X_1 and X_2 are obtained as follows.

Construct W, T, S, R, W_q, $q \in Q$, such that:

- W is a characterisation set of $\mathcal{A}$ (i.e. $\mathcal{A}$ is the specification).
- T is a transition cover of $\mathcal{A}$,
- S is a state cover of $\mathcal{A}$ such that $S \subseteq T$,
- R = T - S
- W_q is a set of sequences that distinguishes q from any other state $q' \in Q$.

Then X_1 and X_2 are defined by:

- $X_1 = S(\Sigma^k W \cup \Sigma^{k-1} W \cup ... \cup W)$,
- $X_2 = \bigcup_{q \in Q} R(\Sigma^k \cup \Sigma^{k-1} \cup ... \cup \{1\}) \otimes W_q$,

where

$$R(\Sigma^k \cup \Sigma^{k-1} \cup ... \cup \{1\}) \otimes W_q = \bigcup_{\substack{s \in R(S^k \cup ... \{1\}) \\ Fe(q_0,s)=q}} \{s\}W_q$$

(i.e. the union is over all $s \in R(\Sigma^k \cup \Sigma^{k-1} \cup ... \cup \{1\})$ such that s takes the machine from the initial state q_0 to q).

Hence $X_2 = \bigcup_{q \in Q} \bigcup_{\substack{s \in R(S^k \cup ... \{1\}) \\ Fe(q_0,s)=q}} \{s\}W_q$.

Intuitively, X_1 checks that all the states defined by the specification are identifiable in the implementation. At the same time, the transitions leading from the initial state to these states are checked for correct output and state transfer. X_2 checks the implementation for all the transitions that are not checked by X_1.

If the sets W_q are all chosen to be W, the characterisation set, then X' = X = TZ, the test set obtained by Chow. Also, since $W_q \subseteq W$, $\forall\, q \in Q$, it follows that $X' \subseteq X$. Therefore the method presented by Fujiwara et al., [16], can yield a smaller test set. Obviously, this is done at the expense of a more complex algorithm to generate the test set.

4.1.4.6. Limitations of the finite state machine testing.

The method enables effective test cases to be generated in a straightforward manner. But the finite state machine is too restrictive for many common applications.

The solution suggested by Chow was to separate the control structure of a program from the data structure and to represent the former as a finite state machine. In this way the method could be used to test the control structure of a program. However, the assumption that the control structure of the system can be modelled separately from the data variables is not realistic in many cases. This would mean that the next state depends solely on the current state and the input. This is not usually the case. The variables that affect the program control could be replaced by a number of additional states, but in many cases this number will be large and then the method would become impractical.

A more attractive solution is to develop testing methods for more complex models than finite state machines.

4.2. Stream X-machine testing.

In Chapter 2 we remarked that stream X-machines possess a property that make them an attractive basis for testing, i.e. the fact that if the basic transition functions are computable by some algorithms (i.e. by finite procedures) than the function computed by the machine can be obtained algorithmically. This is important for two reasons. Firstly, it avoids the unsolvable Halting problem for Turing machines. Secondly, if our specification is a stream X-machine with the processing functions computable by some algorithms, then we are able to determine the output produced for each input sequence. Furthermore, if we know that the implementation can also be modelled by a stream X-machine with the same property, then this implementation is guaranteed to produce an output for any input sequence we apply to it.

However, a testing method for stream X-machines is not straightforward. Indeed, since the machine memory can be infinite, it is fairly clear that there is no way of finding a finite set of input sequences that would guarantee that two arbitrary stream X-machines compute the same function.

The approach we shall use to get around this problem will be a reductionist one. This entails the reduction of a problem to the solution of simpler ones. In such a reductionist approach we would consider a system and produce a testing regime that results in the complete reduction of the test problem for the system to one of looking at the test problem for the components or reduced parts. However, this approach will work only if we are able to make the following statement:

> "the system S is composed of the parts $P_1, ..., P_n$;
> as a result of carrying out a testing process on S we can deduce that S is fault-free if each of $P_1, ..., P_n$ are fault-free".

If the system is a stream X-machine $\mathcal{M}$, then the basic components of the system are the ϕ's. Then, what we are looking for is a testing method that ensures that $\mathcal{M}$ is fault-free provided that Φ is fault-free.

4.2.1. Theoretical basis for stream X-machine testing.

The strategy we employ is to reduce testing that the two machines (i.e. one representing the specification, the other the implementation) compute the same function to testing that their associated automata accept the same language. For this idea to work we require that Φ is complete and output-distinguishable. We call these "design for test" conditions.

The fact that we require Φ to be output-distinguishable is not surprising since we want to be able to distinguish between ϕ's according to the outputs they produce. On the other hand the output-distinguishability condition is not sufficient. Indeed, in Chapter 3 we gave two examples of a stream X-machine with Φ output-distinguishable but not complete in which one of the arcs could be removed without affecting the computation of the machine (see example 3.4.2.7 and example 3.4.2.8).

As we have seen, testing that the two associated automata accept the same language can be done by constructing a set of sequences of elements from Φ^*. However, this is not really very convenient in our case, we really want a set of input sequences from Σ^*. We thus need to convert sequences from Φ^* into sequences from Σ^*. We do this by using a fundamental test function as discussed next.

Definition. 4.2.1.1.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_o, m_o)$ be a deterministic stream X-machine with Φ complete and let $q \in Q$, $m \in M$. We define recursively a function $t_{q,m}: \Phi^* \to \Sigma^*$ as follows:

1. $t_{q,m}(1) = 1$, where 1 is the empty string.

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:

a. if $\exists$ a path $q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \ldots q_{n-1} \xrightarrow{\phi_n} q_n$ in $\mathcal{M}$ starting from q, then

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

where σ_{n+1} is chosen such that

$$(w_e(q, m, t_{q,m}(\phi_1 ... \phi_n)), \sigma_{n+1}) \in \text{dom } \phi_{n+1}.$$

Note: Since Φ is complete, there exists such σ_{n+1}.
In other words, if m_n is the final value computed by the machine along the above path on the input sequence $t_{q,m}(\phi_1 ... \phi_n)$, then (m_n, σ_{n+1}) will exercise ϕ_{n+1}.

b. 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 $\mathcal{M}$ w.r.t. q and m.
If $q = q_o$ and $m = m_o$, $t_{q,m}$ is denoted by t and is called a *fundamental test function* of $\mathcal{M}$.

In other words if

$$q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \ldots q_{n-1} \xrightarrow{\phi_n} q_n$$

is a path in $\mathcal{M}$, then

$$s = t_{q,m}(\phi_1 ... \phi_n)$$

will be an input string which, when applied in q and m, will cause the computation of the machine to follow this path (i.e. $s = \sigma_1 ... \sigma_n$ such that σ_1 exercises ϕ_1, ..., σ_n exercises ϕ_n).
If there is no arc labelled ϕ_{n+1} from q_n, then

$$t_{q,m}(\phi_1 ... \phi_n \phi_{n+1}) = s\sigma_{n+1},$$

where σ_{n+1} is an input which would have caused the machine to exercise such an arc if it had existed (i.e. therefore making sure that it does not exist).

Also, $\forall\ \phi_{n+2}, ..., \phi_{n+k} \in \Phi$,

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

(i.e. therefore only the first non-existing arc in the path is exercised by the value of the test function).

Note that a test function is not uniquely determined, many different possible test functions exist and it is up to the designer to construct it.

Example 4.2.1.2.
Let $\mathcal{M}$ be a deterministic stream X-machine defined by:

1. $\Sigma = \{x, y\}$
2. $\Gamma = \{a, b\}$
3. $Q = \{q_o, q_1, q_2\}$; q_o is the initial state.
4. $M = \{0, 1\}$. The initial memory value is $m_o = 0$.
5. $\Phi = \{\phi_1, \phi_2, \phi_3, \phi_4\}$, where $\phi_1, \phi_2, \phi_3, \phi_4$: $M \times \Sigma \rightarrow \Gamma \times M$ are partial functions as follows:

dom $\phi_1 = M \times \{y\}$;	$\phi_1(m, y) = (a, 1), \forall\ m \in M$;
dom $\phi_2 = M \times \{x\}$;	$\phi_2(m, x) = (a, 0), \forall\ m \in M$;
dom $\phi_3 = M \times \{y\}$;	$\phi_3(m, y) = (b, 1), \forall\ m \in M$;
dom $\phi_4 = M \times \{x\}$;	$\phi_4(m, x) = (b, 0), \forall\ m \in M$.

6. F is represented in figure 4.5.

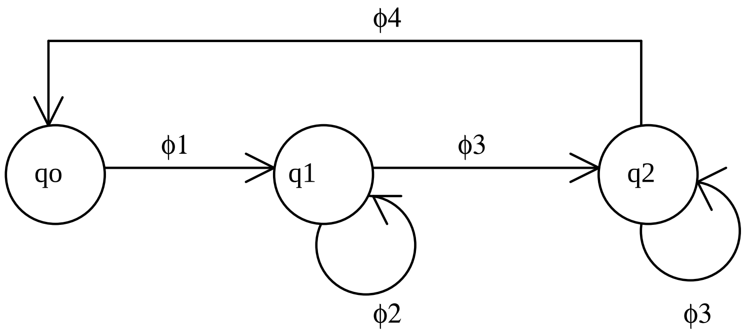


Figure 4.5.

Then we can construct a fundamental test function which satisfies

$t(\phi_1) = y$,

$t(\phi_1\phi_2) = yx$,

$t(\phi_1\phi_2\phi_4) = yxx$,

$t(\phi_1\phi_2\phi_4\phi_1) = yxx$.

The scope of a test function is to test whether a certain path exists or not in $\mathcal{M}$ using appropriate input symbols (hence the name). This idea is formalised in the following lemma.

Lemma 4.2.1.3.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $\mathcal{M}' = (\Sigma, \Gamma, Q', M, \Phi, F', q_0', m_0)$ be two deterministic stream X-machines with Φ output-distinguishable and complete, λ_e and λ_e' their extended output functions and $\mathcal{A}$ and $\mathcal{A}'$ their associated automata. Let $q \in Q$, $q' \in Q'$, $m \in M$, $X \subseteq \Phi^+$, and let $t_{q,m}: \Phi^* \to \Sigma^*$ be a test function of $\mathcal{M}$ w.r.t q and m. If $\lambda_e(q, m, s) = \lambda_e'(q', m, s)$, $\forall\ s \in t_{q,m}(X)$, then q and q' are X-equivalent as states in $\mathcal{A}$ and $\mathcal{A}'$ respectively.

Note: For $X \subseteq \Phi^*$, $t(X) = \{t(x) | x \in X\}$

Proof:
Let $\phi_1 ... \phi_n \in X$ and $s = t_{q,m}(\phi_1 ... \phi_n)$. We prove that $\lambda_e(q, m, s) = \lambda_e'(q', m, s)$ implies:

there exists a path in $\mathcal{M}$ starting from q labelled $\phi_1 ... \phi_n$ iff there exists a path in $\mathcal{M}'$ starting from q' labelled $\phi_1 ... \phi_n$.

Let us assume that there exists a path

$$q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_1 \ldots q_{n-1} \xrightarrow{\phi_n} q_n$$

in $\mathcal{M}$. In this case

$$t_{q,m}(\phi_1 ... \phi_n) = \sigma_1 ... \sigma_n \text{ with } \sigma_1, ..., \sigma_n \in \Sigma.$$

Thus there exist $\gamma_1, \gamma_2, ... \gamma_n \in \Gamma$ and $m_1, ..., m_n \in M$ such that

$$\phi_1(m, \sigma_1) = (\gamma_1, m_1) \text{ and } \phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i),\ i = 2, ..., n.$$

Also, we have

$$\lambda_e(q, m, \sigma_1 ... \sigma_n) = \gamma_1 \gamma_2 ... \gamma_n.$$

Since $\lambda_e(q, m, \sigma_1 ... \sigma_n) = \lambda_e'(q', m, \sigma_1 ... \sigma_n)$, it follows that there exists a path

$$q' \xrightarrow{\phi_1'} q_1' \xrightarrow{\phi_2'} q_2' \ldots q_{n-1}' \xrightarrow{\phi_n'} q_n'$$

in $\mathcal{M}'$ and there exist $m_1', ..., m_n'$ such that

$$\phi_1'(m, \sigma_1) = (\gamma_1, m_1') \text{ and } \phi_i'(m_{i-1}', \sigma_i) = (\gamma_i, m_i'),\ i = 2, ..., n.$$

Using a simple induction, it follows that

$$\phi_i = \phi_i' \text{ and } m_i = m_i',\ i = 1, ..., n.$$

Indeed, $\phi_1 = \phi_1'$ follows since Φ is output-distinguishable. Hence $m_1 = m_1'$. Similarly, if $m_i = m_i'$, it follows that $\phi_{i+1} = \phi_{i+1}'$ and $m_{i+1} = m_{i+1}'$. Therefore, there exists a path in $\mathcal{M}'$ starting from q' labelled $\phi_1 ... \phi_n$.

Let us assume there is no path in $\mathcal{M}$ starting from q labelled $\phi_1 ... \phi_n$. Let $k \in \{0, ..., n-1\}$ be the maximum number such that there exists a path in $\mathcal{M}$ starting from q labelled $\phi_1 ... \phi_k$. Let

$$q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \ldots q_{k-1} \xrightarrow{\phi_k} q_k$$

this path. Then

$$t_{q,m}(\phi_1 ... \phi_n) = \sigma_1 ... \sigma_{k+1} \text{ with } \sigma_1, ..., \sigma_{k+1} \in \Sigma.$$

Thus there exist $\gamma_1, \gamma_2, ..., \gamma_{k+1} \in \Gamma$ and $m_1, ..., m_{k+1} \in M$ such that

$\phi_1(m, \sigma_1) = (\gamma_1, m_1)$ and $\phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i)$, i = 2, ..., k+1.

Now, we prove that there is no path $\mathcal{M}$' starting from q' labelled $\phi_1 ... \phi_{k+1}$. Let us assume otherwise. Then there exists a path

$$q' \xrightarrow{\phi_1} q_1' \xrightarrow{\phi_2} q_2' \ldots q_k' \xrightarrow{\phi_{k+1}} q_{k+1}'$$

in $\mathcal{M}$'. Hence

$\lambda_e'(q', m, \sigma_1 ... \sigma_{k+1}) = \gamma_1 \gamma_2 \cdots \gamma_{k+1}$.

Since $\lambda_e(q, m, \sigma_1 ... \sigma_{k+1}) = \lambda_e'(q', m, \sigma_1 ... \sigma_{k+1})$, it follows that there exists $\phi_{k+1}' \in \Phi$, $q_{k+1} \in Q$ and $m_{k+1}' \in M$ such that

$q_k \xrightarrow{\phi_k'} q_{k+1}$ is an arc in $\mathcal{M}$ and $\phi_{k+1}'(m_k, \sigma_{k+1}) = (\gamma_{k+1}, m_{k+1}')$.

Since Φ is output-distinguishable, it follows that $\phi_{k+1} = \phi_{k+1}'$. This contradicts our initial assumption. Hence, there is no path $\mathcal{M}$' starting from q' labelled

$\phi_1 \cdots \phi_{k+1}$.

Therefore, we have proved that $\lambda_e(q, m, s) = \lambda_e'(q', m, s)$ implies q and q' are $\{\phi_1 ... \phi_n\}$-equivalent as states in $\mathcal{A}$ and $\mathcal{A}$' respectively. Hence (see definition 3.4.1.2.3 and observation 3.4.1.2.4) q and q' are X-equivalent.⑥

We can now assemble our fundamental result which is the basis for the testing method.

Theorem 4.2.1.4.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $\mathcal{M}' = (\Sigma, \Gamma, Q', M, \Phi, F', q_0', m_0)$ be two deterministic stream X-machines with Φ output-distinguishable and complete which compute f and f' respectively, $\mathcal{A}$ and $\mathcal{A}$' their associated automata and t: $\Phi^* \rightarrow \Sigma^*$ a fundamental test function of $\mathcal{M}$. We assume that $\mathcal{A}$ and $\mathcal{A}$' are minimal. Then let T and W, respectively, be a transition cover and a characterisation set of $\mathcal{A}$ and $Z = \Phi^k W \cup \Phi^{k-1} W \cup ... \cup W$, where k is a positive integer. If card(Q') - card(Q) $\leq k$ and f(s) = f'(s), $\forall$ s $\in$ t(TZ), then $\mathcal{A}$ and $\mathcal{A}$' are isomorphic.

Proof:
From lemma 4.2.1.3 it follows that q_0 and q_0' are TZ-equivalent. The rest follows from theorem 4.1.4.1.6 (Chow). ⑥

If our aim is to ensure that the two machines compute the same function the minimality of $\mathcal{A}$' is not really necessary. Then we have the following corollary.

Corollary 4.2.1.5.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $\mathcal{M}' = (\Sigma, \Gamma, Q', M, \Phi, F', q_0', m_0)$ be two deterministic stream X-machines with Φ output-distinguishable and complete, $\mathcal{A}$ and $\mathcal{A}$' the associated automata of $\mathcal{M}$ and $\mathcal{M}$' respectively and t, T and W as above. Let $\mathcal{A}'' = (\Phi, Q'', F'', q_0'')$ be the minimal automaton of $\mathcal{A}$'. If $\mathcal{A}$ is minimal, card(Q'') - card(Q) $\leq k$ and f(s) = f'(s), $\forall$ s $\in$ t(TZ), then f(s) = f'(s) $\forall$ s $\in \Sigma^*$.

Proof:
Let $\mathcal{M}''$ = (Σ, Γ, Q'', M, Φ, F'', q_0'', m_0) the stream X-machine whose associated automaton $\mathcal{A}''$ is the minimal automaton of $\mathcal{A}'$ (i.e. $\mathcal{A}''$ = Min($\mathcal{A}'$)). Then $\mathcal{M}''$ and $\mathcal{M}'$ compute the same function (see lemma 3.4.2.2). From theorem 4.2.1.4, it follows that the associated automata $\mathcal{A}$ and $\mathcal{A}''$ are isomorphic. Hence f = f'.©

Notice that since $\mathcal{A}$ is a finite state machine with empty output alphabet, a characterisation set of $\mathcal{A}$ will be a set $W \subseteq \Phi^*$ such that $\forall$ q, q' $\in$ Q, two states in $\mathcal{A}$ such that $q \neq q'$, there exists $\phi_1 ... \phi_k \in W$ such that either:
there exists a path labelled $\phi_1 ... \phi_k$ from q and there is no path labelled $\phi_1 ... \phi_k$ from q'
or
there exists a path labelled $\phi_1 ... \phi_k$ from q' and there is no path labelled $\phi_1 ... \phi_k$ from q.
In other words, when constructing W what matters is only whether there is an arc labelled with a certain symbol ϕ from a certain state q (since the output alphabet is empty). Obviously, all the statements made in section 4.1.4.3 about the construction of a characterisation set and its cardinality remain valid.

4.2.2. The stream X-machine testing method.

Our stream X-machine testing method is based on the results from theorem 4.2.1.4 and corollary 4.2.1.5.

It assumes that the following conditions are met:

1. The specification is a deterministic stream X-machine $\mathcal{M}$.
2. The set of basic functions **Φ** of $\mathcal{M}$ is output-distinguishable and complete.
3. The associated automaton $\mathcal{A}$ of $\mathcal{M}$ is minimal.
4. The implementation can be modelled as a deterministic stream X-machine $\mathcal{M}'$ with the same set of basic functions Φ; also $\mathcal{M}$ and $\mathcal{M}'$ have the same initial memory value m_0.
5. the number of states of $\mathcal{A}''$, the finite state machine obtained by minimising the associated automaton of $\mathcal{M}'$ (the stream X-machine model of the implementation) is bounded by a certain number, say *n'*.

Then, under these circumstances Y = t(TZ) is a test set that finds *all* faults, where:

- t is a fundamental test function of $\mathcal{M}$,
- T is a transition cover of $\mathcal{A}$,
- W is a characterisation set of $\mathcal{A}$ and
- $Z = \Phi^{k}W \cup \Phi^{k-1}W \cup ... \cup W$,
- $k = n' - n$
- *n* is the number of states of the stream X-machine specification $\mathcal{M}$.
- *n'* is the (estimated) maximum number of states of $\mathcal{A}''$.

First, we remark that in practice the type Φ is always finite. Recall that Φ was defined as the set of (partial) functions that the machine can use (i.e. these may or may not appear in the transition diagram). However, in practice it is natural to restrict Φ to those processing functions that the machine actually uses (i.e. these appear in the transition diagram). Therefore Φ is finite and the test set Y = t(TZ) is also finite.

Obviously, the method relies on the specification being a deterministic stream X-machine. Conditions 2 and 3 lie within the capability of the designer. It is fairly clear that a stream X-machine can be transformed into one with Φ complete and output-distinguishable by adding new inputs and outputs that can be removed after the testing is completed (i.e. these extra inputs and outputs are only used for testing purposes). This issue will be discussed later on together with the possible automation of the process. It is also clear that the designer can arrange for the associated automata of the specification X-machine to be minimal; standard techniques from finite state machine theory are available.

The 4'th condition is the most problematical. Establishing that the set of basic functions, Φ, for the implementation is the same as the specification machine's has to be resolved. In practice this will be done using a separate testing process, depending on the nature of the ϕ's. The method explained above can be applied to test the basic processing functions if they are expressible as the computations of other, simpler X-machines. Alternatively, other testing approaches (e.g. the category partition method or a variant) can be used, if the ϕ's are functions that carry out simple tasks on data structures (i.e. inserting and removing items from registers, stacks, files, i.e.). If the basic processing functions are tried and tested with a long history of successful use (i.e. standard procedures, modules or objects from a library) then their correctness may be accepted.

Once the implementations of the processing functions have been tested, the implementation of the system will consist of:

· the *correct* implementations of the ϕ's;

· 'read' operations; these will be used to read the inputs that will be processed by the ϕ's.

In this case, the implementation will satisfy the 4'th condition if we do not allow two or more ϕ's or two or more read operations to be executed consecutively. However, this technical problem can be overcome easily (e.g. a flag variable can be used to prevent the execution of consecutive ϕ's or read operations; this variable will indicate whether the last piece of code executed was the implementation of a ϕ or a read operations).

Finally, the maximum number of states of the implementation has to be estimated. This is well within the capability of the software developer. In practice k is usually not large (unless there is a considerable degree of misunderstanding on the part of the developer). For critical applications one can make very pessimistic assumptions about k at the expense of a large set. For example, condition 5 could be relaxed to:

5'. The number of states of the stream X-machine model of the implementation is bounded by a number n'.
Then, if the program uses s state variables $v_1, ..., v_s$, then

$$card(Q') \leq card(V_1) \cdot ... \cdot card(V_s),$$

where V_i is the range of v_i, i = 1, ..., n.

A hidden assumption of our method is that a reset operation (i.e. an extra input which causes the machine to change back to the initial state q_o and memory value m_o) is implemented correctly, so that the next test input sequence can be applied from q_o and m_o. In the worst case, this corresponds to restarting the system.

The benefits that accrue if the method is applied is that the entire control structure of the system is tested and *all* faults detected *modulo* the correct implementation of the basic processing functions.

4.2.3. Test set construction

The stream X-machine testing method involves generating the values of a fundamental test function t for all of the sequences in TZ. If the number of inputs is finite (this is always the case in practice) and each transition function ϕ is computable by some algorithm, then the test set Y = t(TZ) can be computed algorithmically. Let us call the algorithms that compute the ϕ's basic algorithms. Then, an algorithm that generates Y will have at most |TZ| steps (i.e. |TZ| denotes the total length of TZ) and each step consists of less then card(Σ) basic algorithms (this is because for $\phi \in \Phi$ and $m \in M$ the algorithm looks for $\sigma \in \Sigma$ such that $(m, \sigma) \in \text{dom } \phi$). If card(Σ) = *p* and card(Φ) = *r*, then the upper bound for the number of basic algorithms applied will be

$$p \cdot \frac{r^{k+2}}{r-1} \cdot n^2 \cdot n' \approx p \cdot r^{k+1} \cdot n^2 \cdot n' \text{ (see section 4.1.4.4)},$$

where *n, n'* and *k* are as defined in the previous section.
However, a more efficient algorithm can be designed if we take advantage of the recursive nature of the test function, i.e. if for a sequence of functions $\phi_1...\phi_i\phi_{i+1}$, $t(\phi_1... \phi_i)$ could be reused in the definition of $t(\phi_1...\phi_i\phi_{i+1})$. This can be done (at least partially) in the procedure shown below.

Let $\mathcal{M}$ = (Σ, Γ, Q, M, Φ, F, q_o, m_o) be a deterministic stream X-machine and $\mathcal{A}$ the associated automaton of $\mathcal{M}$ with

$$Q = \{q_o, q_1,..., q_{n-1}\}$$

and

$$\Phi = \{\phi_1,..., \phi_r\},$$

where Φ is complete. In section 4.1.4.3.1 it is shown that a transition cover T of $\mathcal{A}$ can be constructed using a transition tree. Let

$$T_o = T - \{1\}.$$

Then, by a possible renumbering of Q, T_o can be obtained recursively from the following equalities:

$$T_k = \Phi \cup \bigcup_{\substack{i \in N_k \\ F(q_k, \phi_{ki}) = q_i}} \{\phi_{ki}\} T_i, \ k = 0, ..., n-1, \tag{1}$$

where

$$k_i \in \{1, ..., r\}, \ i \in \bigcup_{k=0}^{n-1} N_k$$

and the sets N_k satisfy the following:

$$N_k \subseteq \{k+1, ..., n-1\}, \ k = 0, ..., n-1, \tag{1.1}$$

$$\bigcup_{k=0}^{n-1} N_k = \{1, ..., n-1\}, \tag{1.2}$$

$$N_i \cap N_j = \varnothing \ \forall \ i, j = 0, ..., n-1, \ i \neq j. \tag{1.3}$$

In other words, the set $\{q_i | \ i \in N_k\}$ is the set of states that can be reached from the state q_k and have not been reached before in the construction of the transition tree; ϕ_{ki} will take the machine from the state q_k to q_i (i.e. $F(q_k, \phi_{ki}) = q_i$).

The equalities above can be rewritten as:

$$T_k = \Phi_k' \cup \Phi_k'' \cup \bigcup_{i \in N_k} \{\phi_{ki}\} T_i, \tag{2}$$

where

$$\Phi_k' = \{\phi \in \Phi | \ F(q_k, \phi) \neq \varnothing\}$$

and

$$\Phi_k'' = \{\phi \in \Phi | \ F(q_k, \phi) = \varnothing\}$$

(i.e. Φ_k' is the set of labels of all the arcs from q_k and $\Phi_k'' = \Phi - \Phi_k'$).

Thus, if we denote $X = TZ$ and $X_k = T_k Z$, $k = 0, ..., n-1$, we have:

$$X = Z \cup X_o \tag{3}$$

and

$$X_k = Z \cup \Phi_k' Z \cup \Phi_k'' Z \cup \bigcup_{i \in N_k} \{\phi_{ki}\} X_i, \ k = 0, ..., n-1. \tag{4}$$

Now let m_o, m_1, ..., $m_{n-1} \in M$, n memory values be defined recursively as follows:

i). m_o is the initial memory value of $\mathcal{M}$.

ii). For j = 1, ..., n-1, m_j is chosen as follows.

Let $k \in \{0, ..., j-1\}$ the (unique) number such that $j \in N_k$ (i.e. k is the unique number such that T_j appears on the right hand side of the equality (1)). The existence and uniqueness of k is ensured by relations (1.2) and (1.3). Since $N_k \subseteq \{k+1, ..., n-1\}$, it also follows that $k < j$.

Hence $F(q_k, \phi_{kj}) = q_j$. Since Φ is complete we can choose $\sigma_j \in \Sigma$ such that $(m_k, \sigma_j) \in \text{dom } \phi_{kj}$. Then we choose m_j such that

$$\phi_{kj}(m_k, \sigma_j) = (\gamma, m_j), \text{ with } \gamma \in \Gamma.$$

Therefore $m_j = w(q_k, m_k, \sigma_j)$ (i.e. m_j is the next memory value after σ_j is applied in q_k and m_k).

Also, let $t_o, t_1, ..., t_{n-1}$ be n test functions that satisfy the following:

i). For $j = 0, ..., n-1$, t_j is a test function w.r.t. q_j and m_j.

ii). For $j = 1, ..., n-1$,

$$t_k(\phi_{kj}) = \sigma_j,$$

where $k \in \{0, ..., j-1\}$ is the (unique) number such that $j \in N_k$ (i.e. this is possible since $(m_k, \sigma_j) \in \text{dom } \phi_{kj}$).

From the way in which $m_o, m_1, ..., m_{n-1}$ and $t_o, t_1, ..., t_{n-1}$ have been chosen it follows that for $k = 0, ..., n-1$ and $j \in N_k$ we have that

$$t_k(\{\phi_{kj}\}V) = \{t_k(\phi_{kj})\}t_j(V) \ \forall\ V \subseteq \Phi^*.$$

Therefore, by applying t_o to (3) and (4) we obtain:

$$t_o(X) = t_o(X_o) \cup t_o(Z) \tag{5}$$

and

$$t_k(X_k) = t_k(\Phi_k' Z) \cup t_k(\Phi_k'' Z) \cup \bigcup_{i \in N_k} \{t_k(\phi_{ki})\}\ t_i(X_i)\ ,\ k = 0, .., n-1. \tag{6}$$

Since $F(q_k, \phi)$ is not defined $\forall\ \phi \in \Phi_k''$, it follows that

$$t_k(\Phi_k'' Z) = t_k(\Phi_k'').$$

Hence (6) becomes:

$$t_k(X_k) = t_k(\Phi_k' Z) \cup t_k(\Phi_k'') \cup \bigcup_{i \in N_k} \{t_k(\phi_{ki})\}\ t_i(X_i)\ ,\ k = 0, ..., n-1. \tag{7}$$

Since $t_k(\phi_{ki}) = \sigma_i$, we obtain:

$$t_k(X_k) = t_k(\Phi_k' Z) \cup t_k(\Phi_k'') \cup \bigcup_{i \in N_k} \{\sigma_i\}\ t_i(X_i)\ ,\ k = 0, ..., n-1. \tag{8}$$

Therefore, a test set $Y = t_o(X)$ can be written as

$$Y = t_o(X_o) \cup t_o(Z),$$

where $t_o(X_o)$ can be obtained recursively from (8).

Example 4.2.3.1.

For the stream X-machine presented in example 4.2.1.2, a transition cover T can be written as

$$T = \{1\} \cup T_o, \text{ where}$$
$$T_o = \{\phi_1\} \cup \{\phi_2, \phi_3, \phi_4\} \cup \{\phi_1\}T_1,$$
$$T_1 = \{\phi_2, \phi_3\} \cup \{\phi_1, \phi_4\} \cup \{\phi_3\}T_2,$$
$$T_2 = \{\phi_3, \phi_4\} \cup \{\phi_1, \phi_2\}$$

The values m_o, m_1 and m_2 are chosen as follows:

$m_o = 0$ is the initial memory value;

$m_1 = 1$ (y takes the machine from q_o and m_o to q_1 and m_1 following the arc labelled ϕ_1);

$m_2 = 1$ (y takes the machine from q_1 and m_1 to q_2 and m_2 following the arc labelled ϕ_3).

Then:

t_o is a test function w.r.t q_o and m_o with $t_o(\phi_1) = y$;
t_1 is a test function w.r.t q_1 and m_1 with $t_1(\phi_3) = y$;
t_2 is a test function w.r.t q_2 and m_2.

Then, a test set $Y = t_o\ (TZ)$ can be written as

$$Y = t_o(Z) \cup t_o(X_o),$$

where

$$t_o(X_o) = t_o(\{\phi_1\}Z) \cup t_o(\{\phi_2, \phi_3, \phi_4\}) \cup \{t_o(\phi_1)\}\ t_1(X_1),$$
$$t_1(X_1) = t_1(\{\phi_2, \phi_3\}Z) \cup t_1(\{\phi_1, \phi_4\}) \cup \{t_1(\phi_3)\}\ t_2(X_2),$$
$$t_2(X_2) = t_2(\{\phi_3, \phi_4\}Z) \cup t_2(\{\phi_1, \phi_2\}).$$

For $W = \{\phi_1, \phi_2\}$ and $n' = n$, we have

$$Z = \{\phi_1, \phi_2\}.$$

Hence, by choosing appropriate values for the test functions (i.e. obviously we require that $t_o(\phi_1) = y$ and $t_1(\phi_3) = y$), the test set Y is:

$$Y = \{y, x\} \cup t_o(X_o), \text{ where}$$
$$t_o(X_o) = \{yy, yx\} \cup \{x, y, x\} \cup \{y\}\ t_1(X_1),$$
$$t_1(X_1) = \{xy, xx, yy, yx\} \cup \{y, x\} \cup \{y\}\ t_2(X_2),$$
$$t_2(X_2) = \{yy, yx, xy, xx\} \cup \{y, x\}.$$

4.2.4. Complexity.

For a stream X-machine with card(Q) = n and card(Φ) = r, the amount of work required to construct W and T is proportional to $r \cdot n^2$ (see section 4.1.4.3.3).

If we generate the test set Y = t(TZ) using the algorithm presented in section 4.2.3, then we have to compute the values of the test function t_o for a domain included in $Z \cup \Phi Z$ and the values of the test functions $t_1, ..., t_{n-1}$ for domains included in ΦZ.

Then, an algorithm that computes t_o will have at most $|Z| + |\Phi Z|$ steps and an algorithm that computes t_i, $1 \leq i \leq n-1$, will have at most $|\Phi Z|$ steps, each step consisting of the following:

· applying the next state function F at most once to obtain the next state;

· applying at most p = card(Σ) basic algorithms to find an appropriate input; also, the next memory value is computed for this input.

Then an upper bound for the total number of basic algorithms applied by the algorithm that generates the test set will be

$$p \cdot (|Z| + n \cdot |\Phi Z|) \approx p \cdot n \cdot |\Phi Z|).$$

The total length of ΦZ, $|\Phi Z|$ can be determined from the following relation:

$$|\Phi Z| = \text{card}(\Phi^{k+1} \cup \Phi^k \cup ... \cup \Phi) \cdot |W| + |\Phi^{k+1} \cup \Phi^k \cup ... \cup \Phi| \cdot \text{card}(W).$$

Since

$$\text{card}(\Phi^{k+1} \cup \Phi^k \cup ... \cup \Phi) = r + r^2 ... + r^{k+1} \le \frac{r^{k+2}}{r-1},$$

$$\text{card}(W) \le n - 1,$$

$$|\Phi^{k+1} \cup \Phi^k \cup ... \cup \Phi| = r + 2 \cdot r^2 ... + (k+1) \cdot r^{k+1} \le \frac{(k+1) \cdot r^{k+2}}{r-1},$$

$$|W| \le \frac{n(n-1)}{2}$$

it follows that

$$|\Phi Z| \le \frac{n(n-1)}{2} \cdot \frac{r^{k+2}}{r-1} + \frac{(k+1) \cdot r^{k+2}}{r-1}(n-1) = \frac{r^{k+2}}{r-1}(n - 1)(k + 1 + \frac{n}{2}) .$$

Therefore, an upper bound for the total number of basic algorithms used in the construction of the test set is approximately

$$p \cdot r^{k+1} \cdot n^2 \cdot (k + \frac{n}{2}) = p \cdot r^{k+1} \cdot n^2 \cdot \frac{2 \cdot n' - n}{2}$$

(this is clearly better then the initial figure we gave in section 4.2.3; in fact, if $n' = n$, the upper bound of the number of basic algorithms is reduced by half).

If the complexity of a basic algorithm is C, then the complexity of the algorithm that generates the test set will be proportional to

$$C \cdot p \cdot r^{k+1} \cdot n^2 \cdot (2 \cdot n' - n).$$

The only problem with this figure is that it depends on p, the number of input symbols, which can be large. However, in practice only some of the inputs will be used in the definition of a particular ϕ, i.e. ϕ will have the form:

$$\phi(m, \sigma) = \begin{cases} \psi(m, \sigma), \text{ if } \sigma \in \Sigma_\phi \\ \emptyset, \text{ if } \sigma \in \Sigma - \Sigma_\phi \end{cases}$$

where Σ_ϕ is a subset of Σ and ψ is a (partial) function ψ: $M \times \Sigma_\phi \rightarrow \Gamma \times M$.
Hence, for $m \in M$, the maximum number of basic algorithms applied for finding σ such that $(m, \sigma) \in \text{dom } \phi$ is card(Σ_ϕ) which may be much lower then card(Σ).

If Φ is complete and output-distinguishable, then the process of generating the test set can be automated.

Similar to the calculations from section 4.1.4.4, the maximum number of test sequences required is less then

$$n^2 \cdot \frac{r^{k+2}}{r-1} \approx n^2 \cdot r^{k+1}$$

and the total length of the test set is less then

$$n^2 \cdot n' \cdot \frac{r^{k+2}}{r-1} \approx n^2 \cdot n' \cdot r^{k+1}.$$

In practice, the total length can be much lower since, in many cases, $|t(\phi_1...\phi_i)| < < i$. Thus test sets generated by the method appear to be of manageable size as is the test application process. If card(Φ) < < card(Σ) (this is usually the case in practice) then the number of test sequences is considerably lower compared with Chow's method.

4.2.5. An improvement in the test set size.

Using the finite state machine theory developed by Fujiwara et al. (see [16] and section 4.1.4.5), it can be shown easily that the test set can be reduced to

$$Y' = t(X'),$$

with

$$X' = X_1 \cup X_2$$

where X_1 and X_2 are obtained as follows.

Let W, T, S, R, W_q, $q \in Q$, such that:

- W is a characterisation set of $\mathcal{A}$ ($\mathcal{A}$ is the associated automaton of the specification),
- T is a transition cover of $\mathcal{A}$,
- S is a state cover of $\mathcal{A}$ such that $S \subseteq T$,
- $R = T - S$,
- W_q is a set of sequences that distinguishes q from any other state $q' \in Q$.

Then X_1 and X_2 are defined by:

- $X_1 = S(\Phi^k W \cup \Phi^{k-1} W \cup \ldots \cup W)$,
- $X_2 = \bigcup_{q \in Q} R(\Phi^k \cup \Phi^{k-1} \cup \ldots \cup \{1\}) \otimes W_q = \bigcup_{q \in Q} \bigcup_{\substack{\upsilon \in R(\Phi^k \cup \ldots \{1\}) \\ Fe(q_0, \upsilon) = q}} \{\upsilon\} W_q$

(i.e. the union is over all $q \in Q$ and $\upsilon \in R(\Phi^k \cup \Phi^{k-1} \cup \ldots \cup \{1\})$ such that υ takes $\mathcal{A}$ from the initial state q_0 to q).

4.2.6. Expected outputs

If Y is the set of input sequences generated using our method, the specification is correct iff $f(s) = f'(s)$, $\forall s \in Y$, where f and f' are the (partial) functions computed by $\mathcal{M}$ and $\mathcal{M}'$ respectively. For many systems f will be a total function, i.e. $\forall s \in \Sigma^*$, the behaviour of the system when it receives the input sequence s is well defined. In this case, the process of comparing the two output sequences 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, if $\mathcal{M}$ is the specification of a software program "f(s) is undefined" would usually mean that the sequence s causes the program to exit.

4.2.7. Imposing "design for test properties" on a system

The stream X-machine testing method we have presented relies on Φ being complete and output-distinguishable. However, if the specification we are dealing with fails to satisfy these requirements, we can augment slightly the basic processing functions using some extra inputs and outputs, such that the resulting specification will satisfy the completeness and output-distinguishability conditions. Obviously, there are many ways in which this can be done. A possible procedure is presented below.

Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ a deterministic stream X-machine with all the states terminal. Then the procedure consists of two steps; the first ensures the completeness and the second the output-distinguishability of the type Φ.

i). Let $\Xi \subseteq \Phi$ be the set of processing functions that are not complete, i.e.

$$\Xi = \{\phi \in \Phi | \ \phi \text{ is not complete}\}.$$

Let $P = \{P_i\}_{i \in I}$ be a partition of Ξ with the following properties:

· $P_i \neq \emptyset, \forall\, i \in I$;

· $\forall\, \phi_1, \phi_2 \in \Xi, \phi_1 \neq \phi_2$, let $i, j \in I$ such that $\phi_1 \in P_i$ and $\phi_2 \in P_j$. If $\exists\, q \in Q$ such that $F(q, \phi_1) \neq \emptyset$ and $F(q, \phi_2) \neq \emptyset$ (i.e. ϕ_1 and ϕ_2 label arcs emerging from the same state), then $i \neq j$.

Let also $\sim_P$ be the equivalence relation induced by P on Ξ.

Let $N_1 = \text{card}(P)$ be the number of elements of the partition P. Let S be a set of new inputs (i.e. $S \cap \Sigma = \emptyset$) with $\text{card}(S) = N_1$ and γ_0 be an output. Let also

$$\zeta: \Xi \rightarrow S$$

be a function that satisfies:

$$\zeta(\phi_1) = \zeta(\phi_2) \text{ iff } \phi_1 \sim_P \phi_2.$$

Then $\forall\, \phi \in \Phi$, we construct a (partial) function

$$\varpi(\phi): M \times (\Sigma \cup S) \rightarrow \Gamma \times M$$

such that:

· if $\phi \in \Phi - \Xi$, then

$\text{dom}\, \varpi(\phi) = \text{dom}\, \phi$ and

$\varpi(\phi)(m, \sigma) = \phi(m, \sigma), \forall\, (m, \sigma) \in \text{dom}\, \phi$;

· if $\phi \in \Xi$, then

$\text{dom}\, \varpi(\phi) = \text{dom}\, \phi \cup (M \times \{\zeta(\phi)\})$ and

$$\varpi(\phi)(m, \sigma) = \begin{cases} \phi(m, \sigma), \text{ if } (m, \sigma) \in \text{dom}\, \phi \\ (\gamma_0, m), \text{ if } \sigma = \zeta(\phi) \text{ and } m \in M \end{cases}$$

It is fairly clear that $\varpi(\phi_1) = \varpi(\phi_2)$ iff $\phi_1 = \phi_2$ and the type $\{\varpi(\phi) | \ \phi \in \Phi\}$ is complete, ii). Let $R = \{R_j\}_{j \in J}$ be a partition of Φ with the following properties:

· $R_j \neq \emptyset\ \forall\ j \in J$;

· $\forall\ \phi_1, \phi_2 \in \Phi$, $\phi_1 \neq \phi_2$, let $i, j \in J$ such that $\phi_1 \in R_i$ and $\phi_2 \in R_j$. If $\varpi(\phi_1)$ and $\varpi(\phi_2)$ are not output-distinguishable, then $i \neq j$.

Let $N_2 = \text{card}(R)$ and let G be a set of outputs with $\text{card}(G) = N_2$. Let also

$$\upsilon: \Phi \rightarrow G$$

be a function that satisfies:

$$\upsilon(\phi_1) = \upsilon(\phi_2) \text{ iff } \phi_1 \sim_R \phi_2,$$

where $\sim_R$ is the equivalence relation induced by R on Φ.

Then $\forall\ \phi \in \Phi$, we construct a (partial) function

$$\rho(\phi): M \times (\Sigma \cup S) \rightarrow (\Gamma \times G) \times M$$

with

$$\text{dom } \rho(\phi) = \text{dom } \varpi(\phi) \text{ and}$$

$$\rho(\phi)(m, \sigma) = ((\gamma, \upsilon(\phi)), m')\ \forall\ (m, \sigma) \in \text{dom } \varpi(\phi),$$

where $(\gamma, m') = \varpi(\phi)(m, \sigma)$.

It is clear that the type

$$\Phi' = \{\rho(\phi) |\ \phi \in \Phi\}$$

is complete and output-distinguishable Also, $\rho: \Phi \rightarrow \Phi'$ is a bijective function.

Then we construct a stream X-machine $\mathcal{M}' = (\Sigma', \Gamma', Q, M, \Phi', F', q_0, m_0)$, where

$\Sigma' = \Sigma \cup S$,

$\Gamma' = (\Gamma \times G)$,

Φ' is defined as above and

$F': Q \times \Phi' \rightarrow Q$ is defined by:

$$F'(q, \phi') = F(q, \rho^{-1}(\phi')), \forall\ q \in Q, \phi' \in \Phi'$$

(i.e. in other words the state transition diagram of $\mathcal{M}'$ is identical to that of $\mathcal{M}$).
The way in which ζ is defined ensures that $\mathcal{M}'$ is deterministic.

The number of extra inputs and outputs required is usually small and the construction of the augmented machine (i.e. $\mathcal{M}'$) from the initial one (i.e. $\mathcal{M}$) is straightforward. In particular, it does not affect the transition diagram, so a transition cover or a characterisation set of the initial machine will still be valid for the augmented one. The initial machine can be obtained from the augmented one by removing the extra inputs and outputs.

Example 4.2.7.1.
Let $\mathcal{M}$ be the following stream X-machine:

1. $\Sigma = \{x, y\}$
2. $\Gamma = \{a, b\}$
3. $Q = \{q_0, q_1, q_2\}$; q_0 is the initial state.
4. $M = \{0, 1\}$. The initial memory value is $m_0 = 0$.

5. $\Phi = \{\phi_1, \phi_2, \phi_3, \phi_4\}$, where $\phi_1, \phi_2, \phi_3, \phi_4$: $M \times \Sigma \rightarrow \Gamma \times M$ are partial function as follows:

$\text{dom } \phi_1 = M \times \Sigma$,
$\phi_1(m, x) = (a, 0)$, $m \in M$,
$\phi_1(m, y) = (a, 1)$, $m \in M$;

$\text{dom } \phi_2 = M \times \{x\}$,
$\phi_2(0, x) = (a, 0)$.
$\phi_2(1, x) = (b, 0)$;

$\text{dom } \phi_3 = \{1\} \times \{y\}$,
$\phi_3(1, y) = (b, 1)$;

$\text{dom } \phi_4 = \{1\} \times \{x\}$,
$\phi_4(1, x) = (b, 0)$.

6. F is represented in figure 4.6.

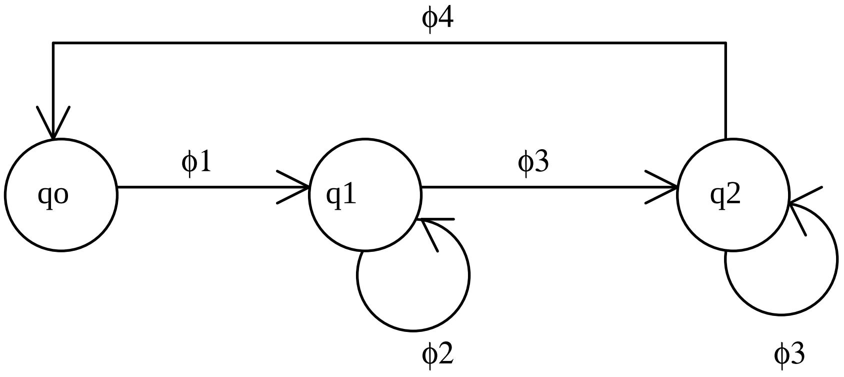


Figure 4.6.

Then
$\Xi = \{\phi_3, \phi_4\}$,
$P = \{\{\phi_3\}, \{\phi_4\}\}$.
We choose $S = \{u, v\}$, $\gamma_o = a$ and we define $\zeta: \Xi \rightarrow S$ by:
$\zeta(\phi_3) = u$, $\zeta(\phi_4) = v$.

Then R can be chosen
$R = \{\{\phi_1, \phi_4\}, \{\phi_2, \phi_3\}\}$.
We also choose $G = \{c, d\}$ and we define $\upsilon: \Phi \rightarrow G_2$ by:
$\upsilon(\phi_1) = \upsilon(\phi_4) = c$; $\upsilon(\phi_2) = \upsilon(\phi_3) = d$.

Then, the augmented machine $\mathcal{M}$' will have
the input alphabet $\Sigma' = \{x, y, z, u\}$,
the output alphabet $\Gamma' = \{a, b\} \times \{c, d\}$ and

$\Phi' = \{\phi_1', \phi_2', \phi_3', \phi_4'\}$, where $\phi_1', \phi_2', \phi_3', \phi_4'$ are partial functions defined by:

$\text{dom } \phi_1' = M \times \{x, y\}$,
$\phi_1'(m, x) = ((a, c), 0), m \in M$,
$\phi_1'(m, y) = ((a, c), 1), m \in M$;

$\text{dom } \phi_2' = M \times \{x\}$,
$\phi_2'(0, x) = ((a, d), 0)$,
$\phi_2'(1, x) = ((b, d), 0)$;

$\text{dom } \phi_3' = (\{1\} \times \{y\}) \cup (M \times \{u\})$,
$\phi_3'(1, y) = ((b, d), 1)$,
$\phi_3'(m, u) = ((a, d), m), m \in M$;

$\text{dom } \phi_4' = (\{1\} \times \{x\}) \cup (M \times \{v\})$,
$\phi_4'(1, x) = ((b, c), 0)$,
$\phi_4'(m, v) = ((a, c), m), m \in M$.

If the procedure above is to be applied successfully then we should be able to determine:

· whether any processing function is complete or not

· whether any two processing functions are output-distinguishable or not.

Usually, this can be done by hand. However, checking these conditions automatically is very difficult since the memory set can be infinite in theory and is usually very large in practice.

However, the above procedure can be relaxed so it can be automated. Before we explain how this can be done, we point out that, if the augmented machine and the test set are constructed by hand, then the number of extra outputs required by the augmented machine could be smaller then that given by the above procedure. First, let us give the following definition.

Definition 4.2.7.2.
Let Φ be a complete type. Then Φ is called *relatively output-distinguishable* if $\forall \phi \in \Phi, m \in M, \exists \sigma \in \Sigma$ such that:

1. $(m, \sigma) \in \text{dom } \phi$; let $(\gamma_1, m_1) = \phi(m, \sigma)$ with $\gamma_1 \in \Gamma, m_1 \in M$.
2. $\forall \phi' \in \Phi, \phi' \neq \phi$, if $\phi'(m, \sigma) = (\gamma_2, m_2)$ with $\gamma_2 \in \Gamma, m_2 \in M$, then $\gamma_1 \neq \gamma_2$.

We also say that the pair (m, σ) *distinguishes* ϕ *in* Φ.

Now, let $\mathcal{M}$ a stream X-machine with Φ complete and relatively output-distinguishable. Then we can restrict the construction of the test function (see definition 4.2.1.1) in the sense that if $t_{q,m}(\phi_1 \dots \phi_{n+1}) = t_{q,m}(\phi_1 \dots \phi_n)\, \sigma_{n+1}$, we require that (m_n, σ_{n+1}) distinguishes ϕ_{n+1} in Φ, where $m_n = w(q, m, t_{q,m}(\phi_1 \dots \phi_n))$. It is easy to see that lemma 4.2.1.3 and theorem 4.2.1.4 remain valid in this case. Therefore, the condition "Φ is complete and output-distinguishable" required by our testing method can be replaced with "Φ is complete and relatively output-distinguishable", provided that the fundamental test function required for the construction of the test set is restricted as shown above. This could lead to a reduction in the number of extra outputs required by the augmented machine. However, in this case, it is very difficult to generate the test set automatically.

Let us now return to the procedure presented at the beginning of this section. This can be transformed into a form suitable for automation by removing the calculations required for checking the completeness and output-distinguishability of the processing functions. Of course, this is done at the expense of a larger number of extra inputs and outputs. The modifications that we make are the following:

i). We take $\Xi = \Phi$.
ii). We require that the partition $R = \{R_j\}_{j\in J}$ satisfies:
· $R_j \neq \emptyset \ \forall \ j \in J$;
· $\forall \ \phi_1, \phi_2 \in \Phi, \phi_1 \neq \phi_2$, let $i, j \in J$ such that $\phi_1 \in R_i$ and $\phi_2 \in R_j$. If $\neg\exists \ q \in Q$ such that $F(q, \phi_1) \neq \emptyset$ and $F(q, \phi_2) \neq \emptyset$, then $i \neq j$.

The rest remains unchanged. The type of the augmented machine will be complete and output-distinguishable (i.e. this is because if ϕ_1 and ϕ_2 are labels of two arcs emerging from the same state, then $\varpi(\phi_1)$ and $\varpi(\phi_2)$ are output-distinguishable since they have disjoint domains).

If we choose P and R to be the partitions with the minimal number of elements, then the upper bounds for the number of inputs and outputs required will be:

$$\text{card}(S) \leq j,$$
$$\text{card}(G) \leq \min(n, r - (i - 1)),$$

where

$$n = \text{card}(Q),\ r = \text{card}(\Phi),$$
$$i = \max_{q\in Q} \text{Card}\{\phi \in \Phi |\ F(q, \phi) \neq \emptyset\},$$
$$j = \max_{\phi\in\Phi} \text{Card}\{\phi' \in \Phi |\ \exists \ q \in Q \text{ such that } F(q, \phi) \neq \emptyset \text{ and } F(q, \phi') \neq \emptyset\}.$$

(i.e. it can be shown easily that $i \leq \text{card}(P) \leq j$; also, since $\text{card}(P) \leq i$, it follows that $\text{card}(R) \leq r - (i - 1)$)

In most cases $j << \text{card}(\Phi)$; hence the number of extra inputs is usually small.

The modified procedure can be automated since it only uses the state transition diagram of the machine to construct the augmented type Φ'.

Example 4.2.7.3.

For the stream X-machine from example 4.2.7.1, we can choose:

$$P = \{\{\phi_1, \phi_3\}, \{\phi_2, \phi_4\}\},\ S = \{u, v\},\ \gamma_0 = a.$$

We define $\zeta: \Xi \rightarrow S$ by:

$\zeta(\phi_1) = \zeta(\phi_3) = u; \quad \zeta(\phi_2) = \zeta(\phi_4) = v.$

We also choose

$R = \{\{\phi_1\}, \{\phi_2\}, \{\phi_3, \phi_4\}\}, G = \{c, d, e\}$

and we define $\upsilon: \Phi \rightarrow G_2$ by:

$\upsilon(\phi_1) = c; \upsilon(\phi_2) = d; \upsilon(\phi_3) = \upsilon(\phi_4) = e.$

Then $\phi_1', \phi_2', \phi_3', \phi_4'$ will be as follows:

$\text{dom } \phi_1' = M \times \{x, y, u\},$
$\phi_1'(m, x) = ((a, c), 0), m \in M,$
$\phi_1'(m, y) = ((a, c), 1), m \in M,$
$\phi_1'(m, u) = ((a, c), m), m \in M;$

$\text{dom } \phi_2' = (\{0, 1\} \times \{x\}) \cup (M \times \{v\}),$
$\phi_2'(0, x) = ((a, d), 0),$
$\phi_2'(1, x) = ((b, d), 0),$
$\phi_2'(m, v) = ((a, d), m), m \in M:$

$\text{dom } \phi_3' = (\{1\} \times \{y\}) \cup (M \times \{u\}),$
$\phi_3'(1, y) = ((b, e), 1),$
$\phi_3'(m, u) = ((a, e), m), m \in M.$

$\text{dom } \phi_4': (\{1\} \times \{x\}) \cup M \times \{u\},$
$\phi_4'(1, x) = ((b, e), 0),$
$\phi_4'(m, v) = ((b, e), m) \; m \in M.$

4.2.8. Case study.

We create a stream X-machine specification of a simplified cash machine. The assumptions we have made are:

- The customer is allowed to enter his personal identification number twice. If both attempts fail, the card is retained.
- Only two fixed sums of money (say £10, £20) can be withdrawn and only one attempt at withdrawing money can be made. If the amount required exceeds the balance of the account, the machine gives an appropriate warning.
- The balance of the account is also available.
- The system does not update the account balance after a transaction has been made. Instead, the new transactions are recorded in a separate data structure and the main data structure is updated at certain time intervals by another system.

4.2.8.1. Stream X-machine specification.

1. The input alphabet is

$$\Sigma = \text{CARDS} \cup \text{STRINGS} \cup \{m\_1, m\_2, b\} \cup \{yes, no\},$$

where:

· CARDS represents the set of all the valid cash cards, (i.e. CARDS = $\{\text{card}_i | \ i = 1 \ldots N\}$).

· STRINGS represents a set of strings of numerals. Each such string is transformed by the machine into a natural number. In practice, only strings of a certain length are allowed.

· {*m_1*, *m_2*, *b*, *yes*, *no*} are distinct inputs that correspond to the options available to the customer; *m_1* and *m_2* correspond to the two amounts of money available and *b* to the balance of the account. *yes* will be used by the customer to request a second service, and *no* to quit the system.

2. The output alphabet is

$$\Gamma = \text{MESSAGES} \times (\text{MONEY} \cup \{null\_m\}) \times (\text{BALANCES} \cup \{null\_b\}) \times \{card\_out, card\_retained, card\_unch\},$$

where:

· MESSAGES = $\{msg_1, ..., msg_{10}, null\_msg\}$, where $msg_1, ..., msg_{10}$ are messages or sequences of messages displayed by the machine as follows:

msg_1 = ’Enter your personal identification number, please.’
msg_2 = 'Would you like: £10, £20, Balance'
msg_3 = 'You have entered a wrong personal identification number. Try again, please.'
msg_4 = 'The card has been retained. □ Insert your card, please.'
msg_5 = 'Would you like another service? yes, no'
msg_6 = 'The amount requested is not available in your account. □ Would you like another service?'
msg_7 = 'Take your card. □ Insert your card, please.'
msg_8 = 'The amount requested is not available in your account. □ Take your card. □ Insert your card, please.'
msg_9 = 'Would you like: Balance'
msg_{10} = 'Would you like: £10, £20'.

· MONEY is a set representing the amounts of money that can be output by the machine; *null_m* denotes that the machine does not output any amount of money;

· BALANCES represent the set of balances; *null_b* denotes that the machine does not output the balance.

· *card_out* denotes that the machine returns the card to the customer; *card_retained* denotes that the card has been retained; *card_unch* denotes that the card state remains unchanged.

3. The memory is

$$M = \text{ACCOUNT\_INFO} \times \text{NEW\_INFO} \times \text{CARD\_NOS},$$

where:

· acc $\in$ ACCOUNT_INFO represents a data structure which contains information concerning each account.

· n_info $\in$ NEW_INFO contains information about the transactions that have been made since the last update.

· CARD_NOS is the set of all possible card numbers. This will include all the valid card numbers and possibly non valid numbers.

4. The initial memory value is

m_0 = (*in_acc*, *in_n_info*, *in_c_no*),

where *in_acc*, *in_n_info* and *in_c_no* are the initial values of ACCOUNT_INFO, NEW_INFO and CARD_NOS.

5. The set of states is:

Q = {Await_card, Await_pin_1, Await_pin_2, Choose_money&balance, Choose_money, Choose_balance, Choose_yes/no_1, Choose_yes/no_2}.

The stream X-machine specification will be a high level one, in the sense that we ignore the way in which ACCOUNT_INFO, NEW_INFO, CARDS, CARD_NOS, MONEY and BALANCES will be represented in the software modelling the system. Instead, we assume that they are manipulated using the following (partial) functions.

Note: B is the set of Booleans; N is the set of positive integers.

· `amount`: {*m_1*, *m_2*} $\rightarrow$ MONEY (injective function)
Retrieves the appropriate amount of money for each of the two options.

· `found`: ACCOUNT_INFO $\times$ CARD_NOS $\rightarrow$ B (function)
Checks whether a certain card number is valid (i.e. whether there is an account that corresponds to this card number).

· `check_account`: ACCOUNT_INFO $\times$ NEW_INFO $\times$ CARD_NOS $\times$ {*m_1*, *m_2*} $\rightarrow$ B (partial function)
Checks whether the amount required is less then the current balance of the account.

· `update_account`: ACCOUNT_INFO $\times$ CARD_NOS $\times$ {*m_1*, *m_2*} $\rightarrow$ NEW_INFO (partial function)
Records the amount of money withdrawn.

· `get_balance`: ACCOUNT_INFO $\times$ CARD_NOS $\rightarrow$ BALANCES
(partial function)
Retrieves the balance of the account.

· `get_card_no`: CARDS $\rightarrow$ CARD_NOS (injective function)
Retrieves the card number

· `get_pin`: ACCOUNT_INFO $\times$ CARD_NOS $\rightarrow$ N (partial function)

Retrieves the personal identification number corresponding to a certain card number.

We also assume that the above functions satisfy:

· $\forall$ acc $\in$ ACCOUNT_INFO c_no $\in$ Im get_card_no, found(acc, c_no);

· dom check_account = ACCOUNT_INFO $\times$ NEW_INFO $\times$ Im get_card_no $\times$ {*m_1*, *m_2*};

· dom update_account = ACCOUNT_INFO $\times$ Im get_card_no $\times$ {*m_1*, *m_2*};

· dom get_balance = ACCOUNT_INFO $\times$ Im get_card_no.

· dom get_pin = ACCOUNT_INFO $\times$ Im get_card_no.

Note: Im f denotes the image of the (partial) function f.

What these conditions say is that for each card there is an account that corresponds to the number of the card and vice versa.

Also, we assume that the function

· convert_string: STRINGS $\rightarrow$ N

converts a string of numerals into a natural number.

6. The type of the machine is:

Φ = {insert_card, enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_balance1, enter_money2, enter_balance2, another_service1, another_service2, no_further_service, ignore_card, ignore_pin, ignore_money, ignore_balance, ignore_options}.

6. The 'next state' function is described in figure 4.7.

7. The basic processing functions are defined as follows.

Note: In what follows acc $\in$ ACCOUNT_INFO, n_info $\in$ NEW_INFO, c_no $\in$ CARD_NOS, card $\in$ CARDS, x $\in$ STRINGS, y $\in$ {*m_1*, *m_2*}.

· dom insert_card: M $\times$ CARDS

insert_card((acc, n_info, c_no), card) =
((*msg*1, *null_m*, *null_b*, *card_unch*), (acc, n_info, get_card_no(card)))

i.e. when the card is inserted, the system reads the card number and the customer is asked to enter his/her personal identification number.

· dom enter_good_pin = {((acc, n_info, c_no), x) $\in$ M $\times$ STRINGS|

`found`(acc, c_no) and `get_pin`(acc, c_no) = `convert_string`(x)}

`enter_good_pin`((acc, n_info, c_no), x) =
    ((*`msg`*2, *`null_m`*, *`null_b`*, *`card_unch`*), (acc, n_info, c_no))

i.e. if the personal identification number is correct, then the customer is allowed to choose one of the following options: two amounts of money and balance.

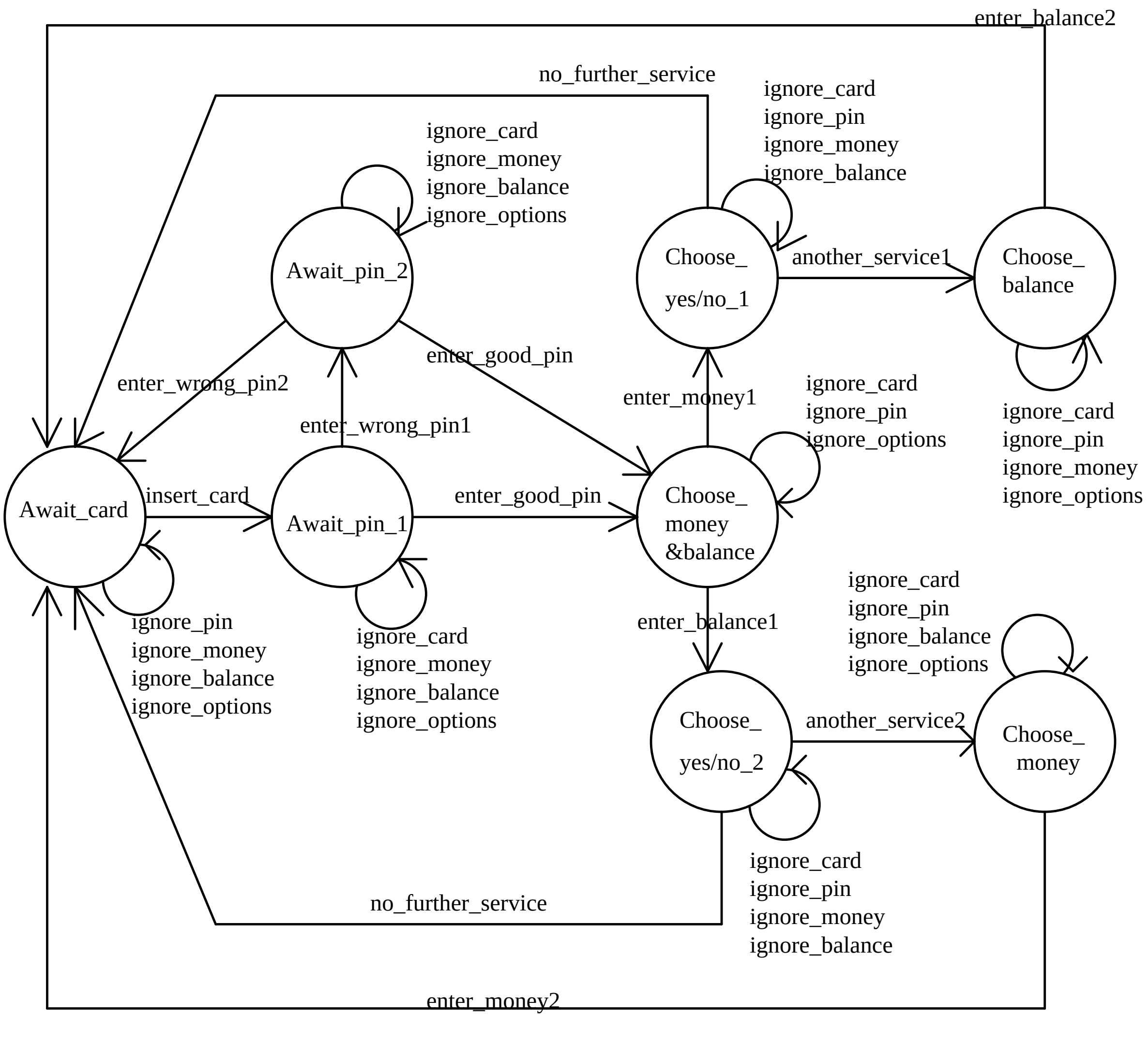


Figure 4.7.

· dom `enter_wrong_pin1` = {((acc, n_info, c_no), x) $\in$ M $\times$ STRINGS| `found`(acc, c_no) and $\neg$(`get_pin`(acc, c_no) = `convert_string`(x))}

`enter_wrong_pin1`((acc, n_info, c_no), x) =
    ((*msg*3, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no))

i.e. if the personal identification number is incorrect, then the customer is asked to enter this again.

· dom `enter_wrong_pin2` = {((acc, n_info, c_no), x) $\in$ M $\times$ STRINGS| `found`(acc, c_no) and $\neg$(`get_pin`(acc, c_no) = `convert_string`(x))}

`enter_wrong_pin2`((acc, n_info, c_no), x) =
    ((*msg*4, *null_m*, *null_b*, *card_retained*), (acc, n_info, c_no)).

i.e. if the customer enters an incorrect personal identification number for the second time, then the card is retained.

· dom `enter_money1` = {(acc, n_info, c_no) $\in$ M| `found`(acc, c_no)} $\times$ {*m_1*, *m_2*}

`enter_money1`((acc, n_info, c_no), y) =
    ((*msg*5, `amount`(y), *null_b*, *card_unch*), (acc, `update_account`(acc, c_no, y), c_no)),
        if `check_account`(acc, n_info, c_no, y) = true
    ((*msg*6, *null_m*, *null_b*, *card_unch*), (acc, n_info, *in_c_no*)),
        if `check_account`(acc, n_info, c_no, y) = false

i.e. the system outputs the required amount of money if this is available in the customer's account and gives a warning message otherwise. The customer is asked whether he/she wants another option.

· dom `enter_money2` = {(acc, n_info, c_no) $\in$ M| `found`(acc, c_no)} $\times$ {*m_1*, *m_2*}

`enter_money2`((acc, n_info, c_no), y) =

((*msg*7, amount(y), *null_b*, *card_out*), (acc, update_account(acc, c_no, y), *in_c_no*)),
if check_account(acc, n_info, c_no, y) = true
((*msg*8, *null_m*, *null_b*, *card_out*), (acc, n_info, *in_c_no*)),
if check_account(acc, n_info, c_no, y) = false

i.e. the system outputs the required amount of money if this is available in the customer's account and gives a warning message otherwise. The customer's card is released, the system returns to the initial state and displays the message 'Insert your card, please'.

· dom enter_balance1 = {(acc, n_info, c_no) $\in$ M| found(acc, c_no)} $\times$ {*b*}

enter_balance1((acc, n_info, c_no), *b*) =
((*msg*5, *null_m*, get_balance(acc, c_no), *card_unch*), (acc, n_info, c_no))

i.e. the system outputs the balance of the account. The customer is asked whether he/she wants another option.

· dom enter_balance2 = {(acc, n_info, c_no) $\in$ M| found(acc, c_no)} $\times$ {*b*}

enter_balance2((acc, n_info, c_no), *b*) =
((*msg*7, *null_m*, get_balance(acc, c_no), *card_out*), (acc, n_info, *in_c_no*))

i.e. the system outputs the balance of the account. The customer's card is released, the system returns to the initial state and displays the message 'Insert your card, please'.

· dom another_service1 = M $\times$ {*yes*}

another_service1((acc, n_info, c_no), *yes*) =
((*msg*9, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no))

i.e. if the "yes" option is chosen, then the "balance" option is displayed.

· dom another_service2 = M $\times$ {*yes*}

another_service2((acc, n_info, c_no), *yes*) =
((*msg*10, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no))

i.e. if the "yes" option is chosen, then the two "money" options are displayed.

· dom `no_further_service` = M × {*no*}

`no_further_service`((acc, n_info, c_no), *no*) =
((*msg*7, *null_m*, *null_b*, *card_out*), (acc, n_info, *in_c_no*))

i.e. if the "no" option is chosen, then the system returns to the initial state and the customer's card is released. The message 'Insert your card, please' is then displayed.

The following functions basically "ignore" a certain input (or number of inputs).

· dom `ignore_card` = M × CARDS

`ignore_card`(((acc, n_info, c_no), card) =
((*null_msg*, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no))

· dom `ignore_pin` = M × STRINGS

`ignore_pin`(((acc, n_info, c_no), x) =
((*null_msg*, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no))

· dom `ignore_money` = M × {*m_1*, *m_2*}

`ignore_money`(((acc, n_info, c_no), y) =
((*null_msg*, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no))

· dom `ignore_balance` = M × {*b*}

`ignore_balance`(((acc, n_info, c_no), *b*) =
((*null_msg*, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no))

· dom `ignore_options` = M × {*yes*, *no*}

`ignore_options`(((acc, n_info, c_no), x) =
((*null_msg*, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no))

4.2.8.2. Imposing the 'design for test' conditions.

The type Φ is output-distinguishable. However, if CARD_NOS includes invalid as well as valid card numbers, then enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_money2, enter_balance1, enter_balance2 are not complete. Let

Ξ = {enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_money2, enter_balance1, enter_balance2}.

Then, we augment the above stream X-machine specification in a similar way (although not an identical way) to the procedure described in section 4.2.7.

Let *new_in1* and *new_in2* be two new inputs (i.e. {*new_in1*, *new_in2*} ∩ Σ = ∅).

Let also *new_msg*1, ..., *new_msg*7 be seven new messages (i.e. {*new_msg*1, ..., *new_msg*7} ∩ MESSAGES = ∅) and let

MESSAGES' = MESSAGES ∪ {*new_msg*1, ..., *new_msg*7}.

The augmented stream X-machine will have the input set

Σ' = Σ ∪ {*new_in1*, *new_in2*}

and the output set

Γ' = MESSAGES' × (MONEY ∪ {*null_m*}) × (BALANCES ∪ {*null_b*}) × {*card_out*, *card_retained*, *card_unch*}.

Then ∀ ϕ ∈ Φ - Ξ, the domain and the definition of ϕ will remain unchanged. For ϕ ∈ Ξ, ϕ will be augmented to ϕ_A using the new inputs and messages. The augmented functions are as follows.

· dom enter_good_pin_A = dom enter_good_pin ∪ (M × {*new_in1*})

enter_good_pin_A((acc, n_info, c_no), x) =
    enter_good_pin((acc, n_info, c_no), x),
        if x ∈ STRINGS
    ((*new_msg*1, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no)),
        if x = *new_in1*

· dom $\text{enter\_wrong\_pin1}_A$ = dom enter_wrong_pin1 ∪ (M × {*new_in2*})

$\text{enter\_wrong\_pin1}_A$((acc, n_info, c_no), x) =
    enter_wrong_pin1((acc, n_info, c_no), x),
        if x ∈ STRINGS
    ((*new_msg*2, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no)),
        if x = *new_in2*

· dom `enter_wrong_pin2`$_A$ = dom `enter_wrong_pin2` $\cup$ (M $\times$ {*new_in2*})

`enter_wrong_pin2`$_A$((acc, n_info, c_no), x) =
  `enter_wrong_pin2`((acc, n_info, c_no), x),
    if x $\in$ STRINGS
  ((*new_msg*3, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no)),
    if x = *new_in2*

· dom `enter_money1`$_A$ = dom `enter_money1` $\cup$ (M $\times$ {*new_in1*})

`enter_money1`$_A$((acc, n_info, c_no), y) =
  `enter_money1`((acc, n_info, c_no), y),
    if y $\in$ {*m_1*, *m_2*}
  ((*new_msg*4, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no)),
    if y = *new_in1*

· dom `enter_money2`$_A$ = dom `enter_money2` $\cup$ (M $\times$ {*new_in1*})

`enter_money2`$_A$((acc, n_info, c_no), y) =
  `enter_money2`((acc, n_info, c_no), y),
    if y $\in$ {*m_1*, *m_2*}
  ((*new_msg*5, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no)),
    if y = *new_in1*

· dom `enter_balance1`$_A$ = dom `enter_balance1` $\cup$ (M $\times$ {*new_in2*})

`enter_balance1`$_A$((acc, n_info, c_no), z) =
  `enter_balance1`((acc, n_info, c_no), z),
    if z = *b*
  ((*new_msg*6, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no)),
    if z = *new_in2*

· dom `enter_balance2`$_A$ = dom `enter_balance2` $\cup$ (M $\times$ {*new_in2*})

`enter_balance2`$_A$((acc, n_info, c_no), z) =
  `enter_balance2`((acc, n_info, c_no), z),
    if z = *b*
  ((*new_msg*7, *null_m*, *null_b*, *card_unch*), (acc, n_info, c_no)),
    if z = *new_in2*

In what follows we shall refer to the augmented versions of these functions.

4.2.8.3. The test set

We generate a test set Y = t(TZ) for the case *n'* - *n* = 0, using the procedure presented in section 4.2.3. First, we construct recursively a transition cover T.

$T = \{1\} \cup T_0$

T_0 = {`insert_card`, `ignore_pin`, `ignore_money`, `ignore_balance`, `ignore_options`} ∪ (Φ - {`insert_card`, `ignore_pin`, `ignore_money`, `ignore_balance`, `ignore_options`}) ∪ {`insert_card`}T_1

T_1 = {`enter_good_pin`, `enter_wrong_pin1`, `ignore_card`, `ignore_money`, `ignore_balance`, `ignore_options`} ∪ (Φ - {`enter_good_pin`, `enter_wrong_pin1`, `ignore_card`, `ignore_money`, `ignore_balance`, `ignore_options`}) ∪ {`enter_wrong_pin1`}T_2 ∪ {`enter_good_pin`}T_3

T_2 = {`enter_good_pin`, `enter_wrong_pin2`, `ignore_card`, `ignore_money`, `ignore_balance`, `ignore_options`} ∪ (Φ - {`enter_good_pin`, `enter_wrong_pin2`, `ignore_card`, `ignore_money`, `ignore_balance`, `ignore_options`})

T_3 = {`enter_money1`, `enter_balance1`, `ignore_card`, `ignore_pin`, `ignore_options`} ∪ (Φ - {`enter_money1`, `enter_balance1`, `ignore_card`, `ignore_pin`, `ignore_options`}) ∪ {`enter_money1`} T_4 ∪ {`enter_balance1`} T_5

T_4 = {`another_service1`, `ignore_card`, `ignore_pin`, `ignore_money`, `ignore_balance`} ∪ (Φ - {`another_service1`, `ignore_card`, `ignore_pin`, `ignore_money`, `ignore_balance`}) ∪ {`another_service1`} T_6

T_5 = {`another_service2`, `ignore_card`, `ignore_pin`, `ignore_money`, `ignore_balance`} ∪ (Φ - {`another_service2`, `ignore_card`, `ignore_pin`, `ignore_money`, `ignore_balance`}) ∪ {`another_service2`} T_7

T_6 = {`enter_balance2` , `ignore_card`, `ignore_pin`, `ignore_money`, `ignore_options`} ∪ (Φ - {`enter_balance2` , `ignore_card`, `ignore_pin`, `ignore_money`, `ignore_options`})

T_7 = {`enter_money2` , `ignore_card`, `ignore_pin`, `ignore_balance`, `ignore_options`} ∪ (Φ - {`enter_money2` , `ignore_card`, `ignore_pin`, `ignore_balance`, `ignore_options`})

A characterisation set is

W = {insert_card, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_money2, another_service1, another_service2}.

Then, for n’ - n = 0, we have

Z = {insert_card, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_money2, another_service1, another_service2}

Let test_card ∈ CARDS be such that

test_no = get_card_no(test_card),

and let test_str1, test_str2 ∈ STRINGS be such that

get_pin(*in_acc*, test_no) = convert_string(test_str1) and

get_pin(*in_acc*, test_no) ≠ convert_string(test_str2).

Let m_0, m, m’ ∈ M be three memory values as follows:

· m_0 = (*in_acc*, *in_n_info*, *in_c_no*) is the initial memory value;

· m = (*in_acc*, *in_n_info*, test_no);

$$m' = \begin{cases} (\mathit{in\_acc}, \texttt{update\_account}(\mathit{in\_acc}, \mathit{m\_1}, \text{test\_no}), \text{test\_no}), \\ \qquad \text{if } \texttt{check\_account}(\mathit{in\_acc}, \text{n\_info}, \text{test\_no}, \mathit{m\_1}) = \text{true} \\ (\mathit{in\_acc}, \mathit{in\_n\_info}, \text{test\_no})), \\ \qquad \text{if } \texttt{check\_account}(\mathit{in\_acc}, \text{n\_info}, \text{test\_no}, \mathit{m\_1}) = \text{false.} \end{cases}$$

Let also t_0, t_1, ..., t_7 be eight test functions as follows:

· t_0 is a test function w.r.t. Await_card and m_0 that satisfies

t_0(insert_card) = test_card;

· t_1 is a test function w.r.t. Await_pin_1 and m that satisfies

t_1(enter_good_pin) = test_str1,

t_1(enter_wrong_pin1) = test_str2;

· t_2 is a test function w.r.t. Await_pin_2 and m;

· t_3 is a test function w.r.t. Choose_money&balance and m that satisfies

t_3(enter_money1) = *m_1*,

t_3(enter_balance1) = *b*;

· t_4 is a test function w.r.t. Choose_yes/no_1 and m’ that satisfies

t_4(another_service1) = *yes*;

· t_5 is a test function w.r.t. Choose_yes/no_2 and m that satisfies

t_5(another_service2) = *yes*;

· t_6 is a test function w.r.t. Choose_balance and m’.

· t_7 is a test function w.r.t. Choose_money and m.

Then, a test set t_0(X), X = TZ, can be written as (see section 4.2.3):

$t_0(X) = t_0(Z) \cup t_0(X_0)$,
where

$t_0(X_0) = t_0($\{insert_card, ignore_pin, ignore_money, ignore_balance, ignore_options\} Z) $\cup$ $t_0($\{enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_money2, enter_balance1, enter_balance2, another_service1, another_service2, no_further_service, ignore_card\}) $\cup$ \{t(insert_card)\} $t_1(X_1)$

$t_1(X_1) = t_1($\{enter_good_pin, enter_wrong_pin1, ignore_card, ignore_money, ignore_balance, ignore_options\} Z) $\cup$ $t_1($\{insert_card, enter_wrong_pin2, enter_money1, enter_money2, enter_balance1, enter_balance2, another_service1, another_service2, no_further_service, ignore_pin\}) $\cup$ \{t_1(enter_wrong_pin1)\} $t_2(X_2)$ $\cup$ \{t_1(enter_good_pin)\} t_3 (X_3)

$t_2(X_2) = t_2($\{enter_good_pin, enter_wrong_pin2, ignore_card, ignore_money, ignore_balance, ignore_options\} Z) $\cup$ $t_2($\{insert_card, enter_wrong_pin1, enter_money1, enter_money2, enter_balance1, enter_balance2, another_service1, another_service2, no_further_service, ignore_pin\})

$t_3(X_3) = t_3$ (\{enter_money1, enter_balance1, ignore_card, ignore_pin, ignore_options\} Z) $\cup$ t_3 (insert_card_card, enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money2, enter_balance2, another_service1, another_service2, no_further_service, ignore_money, ignore_balance) $\cup$ \{t_3 (enter_money1)\} t_4 (X_4) $\cup$ \{t_3(enter_balance1)\} $t_5(X_5)$

$t_4(X_4) = t_4($\{another_service1, no_further_service, ignore_card, ignore_pin, ignore_money, ignore_balance\} Z) $\cup$ $t_4($\{insert_card, enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_money2, enter_balance1, enter_balance2, another_service2, ignore_options\}) $\cup$ \{t_4(another_service1)\} $t_6(X_6)$

$t_5(X_5) = t_5($\{another_service2, no_further_service, ignore_card, ignore_pin, ignore_money, ignore_balance\} Z) $\cup$ $t_5($\{insert_card, enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_money2, enter_balance1, enter_balance2, another_service1, ignore_options\}) $\cup$ \{t_5(another_service2)\} $t_7(X_7)$

$t_6(X_6) = t_6${enter_balance2, ignore_card, ignore_pin, ignore_money, ignore_options} Z) $\cup$ t_6({insert_card, enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_money2, enter_balance1, another_service1, another_service2, no_further_service, ignore_balance})

$t_7(T_7) = t_7$({enter_money2, ignore_card, ignore_pin, ignore_balance, ignore_options} Z) $\cup$ t_7({insert_card, enter_good_pin, enter_wrong_pin1, enter_wrong_pin2, enter_money1, enter_balance1, enter_balance2, another_service1, another_service2, no_further_service, ignore_money})

Let us assume that *in_c_no* is not a valid card number (i.e. *in_c_no* $\notin$ Im get_card_no). Then, by choosing suitable values for the test functions t_o, ..., t_7 the test set becomes:

$t_o(X)$ = {test_card, *new_in2*, *new_in2*, *new_in1*, *new_in1*, *yes*, *yes*} $\cup$ $t_o(X_o)$

$t_o(X_o)$ = {test_card test_card, test_card test_str2, test_card test_str2, test_card *m_1*, test_card *m_1*, test_card *yes*, test_card *yes*, test_str1 test_card, test_str1 *new_in2*, test_str1 *new_in2*, test_str1 *new_in1*, test_str1 *new_in1*, test_str1 *yes*, test_str1 *yes*, *m_1* test_card, *m_1* *new_in2*, *m_1* *new_in2*, *m_1* *new_in1*, *m_1* *new_in1*, *m_1* *yes*, *m_1* *yes*, *b* test_card, *b* *new_in2*, *b* *new_in2*, *b* *new_in1*, *b* *new_in1*, *b* *yes*, *b* *yes*, *yes* test_card, *yes* *new_in2*, *yes* *new_in2*, *yes* *new_in1*, *yes* *new_in1*, *yes* *yes*, *yes* *yes*} $\cup$ {*new_in1*, *new_in2*, *new_in2*, *new_in1*, *new_in1*, *new_in2*, *new_in2*, *yes*, *yes*, *no*, test_card} $\cup$ {test_card} $t_1(X_1)$

$t_1(X_1)$ = {test_str1 test_card, test_str1 test_str2, test_str1 test_str2, test_str1 *m_1*, test_str1 *m_2*, test_str1 *yes*, test_str1 *yes*, test_str2 test_card, test_str2 test_str2, test_str2 test_str2, test_str2 *m_1*, test_str2 *m_1*, test_str2 *yes*, test_str2 *yes*, test_card test_card, test_card test_str2, test_card test_str2, test_card *m_1*, test_card *m_1*, test_card *yes*, test_card_*yes*, *m_1* test_card, *m_1* test_str2, *m_1* test_str2, *m_1* *m_1*, *m_1* *m_1*, *m_1* *yes*, *m_1* *yes*, *b* test_card, *b* test_str2, *b* test_str2, *b* *m_1*, *b* *m_1*, *b* *yes*, *b* *yes*, *yes* test_card, *yes* test_str2, *yes* test_str2, *yes* *m_1*, *yes* *m_1*, *yes* *yes*, *yes* *yes*} $\cup$ {test_card, test_str2, *m_1*, *m_1*, *b*, *b*, *yes*, *yes*, *no*, test_str1} $\cup$ {test_str2} $t_2(T_2)$ $\cup$ {test_str1} $t_3(T_3)$

$t_2(X_2)$ = {test_str1 test_card, test_str1 test_str2, test_str1 test_str2, test_str1 *m_1*, test_str1 *m_1*, test_str1 *yes*, test_str1 *yes*, test_str2 test_card, test_str2 test_str2, test_str2 test_str2, test_str2 *m_1*, test_str2 *m_1*, test_str2 *yes*, test_str2 *yes*, test_card test_card, test_card test_str2, test_card test_str2, test_card *m_1*, test_card *m_1*, test_card *yes*, test_card_*yes*, *m_1* test_card, *m_1* test_str2, *m_1* test_str2, *m_1* *m_1*, *m_1* *m_1*, *m_1* *yes*, *m_1* *yes*, *b* test_card, *b* test_str2, *b* test_str2, *b m_1*, *b m_1*, *b yes*, *b yes*, *yes* test_card, *yes* test_str2, *yes* test_str2, *yes m_1*, *yes m_1*, *yes yes*, *yes yes*} ∪ {test_card, test_str2, *m_1*, *m_1*, *b*, *b*, *yes*, *yes*, *no*, test_str1}

$t_3(X_3)$ = {*m_1* test_card, *m_1* test_str2, *m_1* test_str2, *m_1 m_1*, *m_1 m_1*, *m_1 yes*, *m_1 yes*, *b* test_card, *b* test_str2, *b* test_str2, *b m_1*, *b m_1*, *b yes*, *b yes*, test_card test_card, test_card test_str2, test_card test_str2, test_card *m_1*, test_card *m_1*, test_card *yes*, test_card *yes*, test_str1 test_card, test_str1 test_str2, test_str1 test_str2, test_str1 *m_1*, test_str1 *m_1*, test_str1 *yes*, test_str1 *yes*, *yes* test_card, *yes* test_str2, *yes*_test_str2, *yes m_1*, *yes m_1*, *yes yes*, *yes yes*} ∪ {test_card, test_str1, test_str2, test_str2, *m_1*, *b*, *yes*, *yes*, *no*, *m_1*, *b*} ∪ {*m_1*} $t_4(X_4)$ ∪ {*b*} $t_5(X_5)$

$t_4(X_4)$ = {*yes* test_card, *yes* test_str2, *yes* test_str2, *yes m_1*, *yes m_1*, *yes yes*, *yes yes*, *no* test_card, *no new_in2*, *no new_in2*, *no new_in1*, *no new_in1*, *no yes*, *no yes*, test_card test_card, test_card test_str2, test_card test_str2, test_card *m_1*, test_card *m_1*, test_card *yes*, test_card *yes*, test_str1 test_card, test_str1 test_str2, test_str1 test_str2, test_str1 *m_1*, test_str1 *m_1*, test_str1 *yes*, test_str1 *yes*, *m_1* test_card, *m_1* test_str2, *m_1* test_str2, *m_1 m_1*, *m_1 m_1*, *m_1 yes*, *m_1 yes*, *b* test_card, *b* test_str2, *b* test_str2, *b m_1*, *b m_1*, *b yes*, *b yes*} ∪ {test_card, test_str1, test_str2, test_str2, *m_1*, *m_1*, *b*, *b*, *yes*, *yes*} ∪ {*yes*} $t_6(X_6)$

$t_5(X_5)$ = {*yes* test_card, *yes* test_str2, *yes* test_str2, *yes m_1*, *yes m_1*, *yes yes*, *yes yes*, *no* test_card, *no new_in2*, *no new_in2*, *no new_in1*, *no new_in1*, *no yes*, *no yes*, test_card test_card, test_card test_str2, test_card test_str2, test_card *m_1*, test_card *m_1*, test_card *yes*, test_card *yes*, test_str1 test_card, test_str1 test_str2, test_str1 test_str2, test_str1 *m_1*, test_str1 *m_1*, test_str1 *yes*, test_str1 *yes*, *m_1* test_card, *m_1* test_str2, *m_1* test_str2, *m_1 m_1*, *m_1 m_1*, *m_1 yes*, *m_1 yes*, *b* test_card, *b* test_str2, *b* test_str2, *b m_1*, *b m_1*, *b yes*, *b yes*} ∪ {test_card, test_str1, test_str2, test_str2, *m_1*, *m_1*, *b*, *b*, *yes*, *yes*} ∪ {*yes*} $t_7(X_7)$

$t_6(X_6)$ = {*b* test_card, *b new_in2*, *b new_in2*, *b new_in1*, *b new_in1*, *b yes*, *b yes*, test_card test_card, test_card test_str2, test_card test_str2, test_card *m_1*, test_card *m_1*, test_card *yes*, test_card *yes*, test_str1 test_card, test_str1 test_str2, test_str1 test_str2, test_str1 *m_1*, test_str1 *m_1*, test_str1 *yes*, test_str1 *yes*, *m_1* test_card, *m_1* test_str2, *m_1* test_str2, *m_1 m_1*, *m_1 m_1*, *m_1 yes*, *m_1 yes*, *yes* test_card, *yes* test_str2, *yes* test_str2, *yes m_1*, *yes m_1*, *yes yes*, *yes yes*} ∪ {test_card, test_str1, test_str2, test_str2, *m_1*, *m_1*, *b*, *yes*, *yes*, *no*, *b*}

$t_7(X_7)$ = {*m_1* test_card, *m_1 new_in2*, *m_1 new_in2*, *m_1 new_in1*, *m_1 new_in1*, *m_1 yes*, *m_1 yes*, test_card test_card, test_card test_str2, test_card test_str2, test_card *m_1*, test_card *m_1*, test_card *yes*, test_card *yes*, test_str1 test_card, test_str1 test_str2, test_str1 test_str2, test_str1 *m_1*, test_str1 *m_1*, test_str1 *yes*, test_str1 *yes*, *b* test_card, *b* test_str2, *b* test_str2, *b m_1*, *b* *m_1*, *b yes*, *b yes*, *yes* test_card, *yes* test_str2, *yes* test_str2, *yes m_1*, *yes m_1*, *yes yes*, *yes yes*} ∪ {test_card, test_str1, test_str2, test_str2, *m_1*, *b*, *b*, *yes*, *yes*, *no*, *m_1*}

4.2.8.4. Discussion.

The success of our testing procedure relies on the basic functions Φ being correctly implemented. So, before we apply our testing methods we have to ensure that this is the case. Therefore, the testing process can be viewed as a process consisting of three stages:

1. Testing the functions that manipulate the inputs and the data structure of the system (i.e. `convert_string`, `check_account`, `update_account`, `get_card_no`, etc.). A prerequisite is that these functions have to be clearly specified (preferably a formal specification). In practice these functions perform fairly standard operations on common data structures (i.e. add an item to a file, retrieve an item from a file, convert a string into a positive integer, etc.). Usually, these are standard routines and can be assumed to be fault-free. If this is not the case, then category-partition testing can be used.
2. Testing the ϕ's. These are very simple functions obtained in a straightforward manner from the functions above and a simple category partition testing can be successfully used.
3. Testing the control structure of the system. We use the test set generated above.

4.2.9. Generalised stream X-machine testing.

We now consider how our method can be applied to generalised stream X-machines. Obviously, the definitions of completeness, output-distinguishability and that of a test function can be extended in a straightforward manner to generalised stream X-machines. There is however a problem in the sense that the proof of lemma 4.2.1.3 - and hence the proof of theorem 4.2.1.4 - relies on the following property of stream functions:

If f, f':$\Sigma^* \to \Gamma^*$ are (partial) stream functions then: $\forall$ x, y $\in \Sigma^*$, if

$$f(xy) = f'(xy) \neq \emptyset,$$

then

$$f(x) = f'(x).$$

This is not true for generalised stream functions.

We shall now discuss how this problem can be addressed. First, let us give the following definitions.

Definition 4.2.9.1.
Let Σ an alphabet and x $\in \Sigma^*$. Then y $\in \Sigma^*$ is called a *prefix of* x if $\exists$ z $\in \Sigma^*$ such that x = yz. We also define

$$Pref(x) = \{y \in \Sigma^* \mid y \text{ is a prefix of } x\}$$

the set of all prefixes of x.

Definition 4.2.9.2.
Let Σ an alphabet and $X \subseteq \Sigma^*$. Then we define the set *Pref*(X) by:

$$\mathrm{Pref}(X) = \bigcup_{x \in X} \mathrm{Pref}(x).$$

Now, if replace the test set t(TZ) with Y = Pref(t(TZ)) theorem 4.2.1.4 is also true for generalised stream X-machines. Therefore, if our specification is a generalised stream X-machine, then Y = Pref(t(TZ)) is a test set that finds all the faults in the implementation, provided that the conditions from section 4.2.2 are met.

At first sight, it appears that generalised stream X-machines require much larger test sets. However, this is not really the case since, if a system receives an input sequence s, it also receives all the prefixes of that sequence. Therefore, the test set t(TZ) can be also used for generalised stream X-machines provided that results are recorded in a way that allows the outputs produced by all prefixes of the sequences in t(TZ) to be determined. For example, these results can have the form

$\{\sigma_1/g_1 ..\sigma_i/g_i | \sigma_1...\sigma_i \in t(TZ)\}$,

where g_j, $j \leq i$, is the output sequence corresponding to σ_j.

4. 2.10. (Generalised) stream X-module testing.

Our testing method does not depend on the initial memory of the machine. What we are really testing is that the associated automata of the two machines, one representing the specification and the other the implementation, accept the same language. Therefore, if we change the initial memory of both the specification and the implementation, they will still compute the same function (of course, as long as the initial memory of the specification coincides with that of the implementation). This idea is formalised in what follows.

A (generalised) stream X-machine with unspecified initial memory will be called a (generalised) stream X-module.

Definition 4.2.10.1.
A *(generalised) stream X-module* is a tuple $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, T)$, where Σ, Γ, Q, M, Φ, F, q_0, T have the same meanings as for stream X-machines.

A stream X-module can be regarded as the set of all the stream X-machines $\mathcal{M}_{m_0} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, T, m_0)$ with m_0 taking all the values in M. As for deterministic stream X-machines, a deterministic (generalised) stream X-module is one in which Φ is a set of partial functions, F is a partial function $F: Q \times \Phi \to Q$ and any two ϕ's that are used as labels of arcs emerging from the same state have disjoint domains.

Similar to (generalised) stream X-machine, we can define the transition function u, the output function λ, the extended transition function u_e and the extended output function λ_e for (generalised) stream X-modules. Also, the definition of the associated automaton of a module is identical to that of a machine.

Definition 4.2.10.2.
Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, T)$ be a (generalised) stream X-module and let $f: M \times \Sigma^* \to \Gamma^*$ be a (partial) function. Then we say that $\mathcal{M}$ computes f iff:

$\forall m_0 \in M, s \in \Sigma^*, f(m_0, s) = f_{m_0}(s)$,

where f_{m_0} is the (partial) function computed by the machine $\mathcal{M}_{m_0} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, T, m_0)$ (i.e. the initial memory is m_0).

As for X-machines, we shall assume that the modules we shall be referring to are modules with all the states terminal. Then, our testing method can be extended to (generalised) stream X-modules. In this case, a fundamental test function will be a function w.r.t. q_0, the initial state of the module, and any memory value $m \in M$.

4.2.11. Discussion and conclusions.

The main benefit of the stream X-machine testing method is that if the implementation passes all of the tests in the test set then it is known to be free of faults, providing that the ϕ's have been implemented in a fault-free fashion. That is, we can replace the problem of testing for all faults in a stream X-machine to one of detecting faults in something simpler - namely the processing functions. These ϕ's usually fall into one of the following categories.

1). A processing function is a (simpler) stream X-machine itself (an example that illustrates this idea is given in section 5.4.5). For very complex systems, the ϕ's can be stream X-machines whose processing functions are stream X-machines themselves. In this way, the reduction process has more than one level and the stream X-machine testing method is applied to each of these levels.

2). In many cases the processing functions are very straightforward functions that carry out simple operations on common data structures (i.e. files, stacks, etc.) or are simple arithmetic operations, or process character strings, etc. Usually, these are standard routines from a library and they can be assumed to be fault-free. In the worst case, a category-partition like method can be successfully used to test this type of function.

3). A ϕ is obtained using a composition of functions of the type described in 1) and 2) and possibly some other simple functions (e.g. an 'if - else' statement containing functions of the type 1) and 2)) In this case, a category-partition method will usually be sufficient, provided that the lower-level functions (i.e. those of type 1) and 2)) have already been tested.

The test set produced by the stream X-machine method is of manageable size (it depends polynomially on the number of states and the number of processing functions of the machine). The process of generating the test set can be automated, the complexity of the algorithm being proportional to the complexity of the algorithms that compute the ϕ's .