A method for refining and testing generalised machine specifications

Florentin Ipate & Mike Holcombe

Formal Methods and Software Engineering (FORMSOFT) Group

Department of Computer Science

University of Sheffield,

Regent Court, 211, Portobello Street

Sheffield S1 4DP, UK.

Correspondence to:

Professor Mike Holcombe

Department of Computer Science

University of Sheffield,

Regent Court, 211, Portobello Street

Sheffield S1 4DP, UK.

Email : M.Holcombe@dcs.shef.ac.uk

Abstract.

Two current areas of emphasis in the development of methods for engineering higher quality and safer software are the use of formal methods for the specification and verification of software and the development of sophisticated methods of software and system testing. In general these two approaches are unrelated and few examples exist whereby the best aspects of both are used in an integrated way. Ipate and Holcombe [6] present such an integrated approach by using a type of generalised machine, namely the stream X-machine, both as a specification tool and as a basis of a testing method. However, if this type of specification method is to become usable in a wide range of software applications and acceptable to a wide community of software engineers then there needs to be ways of refining existing specifications into more complex and more detailed versions. Also, for each such process of refinement, there needs to be methods of deriving the test set for the resulting machine from that of the initial machine. This paper present such a refinement and also provides a method for testing machines constructed through such a process of refinement.

Key words: testing, test set, formal specification, finite state machines, X-machines, refinement.

1. Introduction.

The subject of software quality, of delivering a *correct* implementation of the *correct* system, has occupied the attention of many software engineers and generated a substantial literature. Two main areas of emphasis in the development of higher quality software is the use of formal methods for the specification and verification of software and the development of methods of software and system testing. Unfortunately, very often these two areas are regarded as mutually exclusive and very few attempts have been made to bridge the gap between them. Many within the formal methods community pour scorn on the whole concept of testing and insist that all a test can tell us is that the system has failed - it cannot tell us that the system is correct. This is true of most testing methods currently in use and is clearly a major drawback. Most testing methods are practice-driven approaches rather than carefully constructed engineering methods based on a defensible well-founded strategy, so they do not allow us to make any statements about the type and the number of faults that remain in the implementation after testing is completed. In particular, they cannot insure that the system is *fault-free* once it has passed the testing process. However, it is hardly credible to expect that any complete system would be released without substantial testing, especially since the alternative to this, formal verification, has not been able to demonstrate yet that it is "scalable" and practical in the context of the massive complexity of today's applications.

A possible answer to the problem of software quality would be the development of an integrated formal specification and testing methodology. Indeed, by considering testing from the specification phase, we shall produce formal specifications that are more easily

testable. Conversely, by developing a testing theory based on a particular formal model, we can place testing on a theoretical basis, thereby providing a more convincing approach to the problem of detecting *all* faults and allowing us to make sensible and theoretically defensible claims about the level and type of faults that remain in the implementation after testing is completed.

Ipate and Holcombe [6] set the basis for such an integrated methodology by considering testing based on an algebraic approach to computational modelling. They use a type of generalised machine, namely the stream X-machine, both as a specification tool and as a basis of a testing method. The clear advantage of their testing method is that the test set generated by it is guaranteed to detect *all faults* in the specification provided that some initial conditions are satisfied. These can be either imposed by the designer or can be ensured using a separate testing process. Furthermore, the specification method used, the X-machine, proves to be intuitive and easy to use as well as general enough to cope with a wide range of applications, see Holcombe [3], Ipate[5].

However, if the X-machine is to become a basis for an integrated specification and testing method which can be used widely in practice there needs to be ways of refining existing specifications into more complex and more detailed versions. Also, for each such process of refinement, there needs to be methods of deriving the test set for the resulting machine from that of the initial machine. In this paper we make a step in this direction by introducing the concept of *refinement* as a way of developing stream X-machine

specifications gradually. A method for testing machines constructed through such a process of refinement is also given.

2. Refinement - definitions.

The concept of refinement we are presenting here provides a way of constructing more complex X-machine specifications from simpler machines. The way in which these ‘basic’ machines are joined together is specified by a 'control' machine. The situation is similar to that of an implementation that uses a 'main' program that calls several sub-programs.

Before we proceed any further, let us introduce the concept of *stream X-module* as a stream X-machine with unspecified initial memory.

Definition 2.1.

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

A stream X-module can be regarded as the set of 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 stream X-module is one in which Φ is a

set of partial functions, F is a partial function $F: Q \times \Phi \rightarrow Q$ and any two ϕ's that are used as labels of arcs emerging from the same state have disjoint domains (see Ipate and Holcombe [6] or Ipate [5]).

Similar to 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 stream X-modules. Also, the definition of the associated automaton of a module is identical to that of a machine. As for X-machines, we shall assume that the modules we shall be referring to are modules with all the states terminal.

Let us explain now how our refinement works.

Let Σ be an input alphabet and Γ_1, Γ_2 two output alphabets. Let

$$\mathcal{M}_i = (\Sigma, \Gamma_1, P_i, M_i, \Phi_i, F_i, p_{io}), \ i = 0, ..., n,$$

be deterministic stream X-modules (P_i is the state set, p_{io} is the initial state and all states are assumed to be terminal). For i = 0, ..., n we assume that the following conditions are satisfied:

i) The associated automata of the modules $\mathcal{M}_i$ are minimal.

ii) There exists a state $p_{if} \in P_i$, $p_{io} \neq p_{if}$, such that $\forall \ \phi_i \in \Phi_i$, $F_i(p_{if}, \phi_i) \neq \emptyset$ (i.e. no arcs emerge from the state p_{if}).

We shall call p_{if} the *final state* of the module (i.e. the name indicates that once the module is in p_{if} no further transitions are allowed).

Note: The notion of final state should not be confused with that of terminal state.

iii) Also, for the sake of simplicity we assume that all the transitions that lead to the final state p_{if} output a special symbol $\delta \in \Gamma_1$ called *end marker*. More formally, this can be written as:

$$\forall\ \phi_i \in \Phi_i, \text{ if } \exists\ p_i \in P_i - \{p_{if}\} \text{ such that } F(p_i, \phi_i) = p_{if} \text{ then Im } \phi_i \subseteq M_i \times \{\delta\}.$$

Obviously, the end marker has no corresponding physical representation in the system specified by the machine $\mathcal{M}_i$. It is only used as a way to indicate that the system has reached its end state.

The modules $\mathcal{M}_i$ will be called the *refinement modules*.

Let

$$\mathcal{M} = (I, \Gamma_2, Q, M, \Phi, F, q_o, m_o)$$

be a deterministic stream X-machine with the state set $Q = \{q_o, q_1,... q_n\}$, the input set I, the output set Γ_2 and the memory set M and let

$$z_i: M \rightarrow M_i,\ y_i: M_i \rightarrow I,\ i = 0,..., n,$$

be functions.

$\mathcal{M}$ will be called the *control* machine. This will be refined using the modules $\mathcal{M}_i$. In fact, all the arcs that emerge from the state q_i will be refined by the module $\mathcal{M}_i$. When $\mathcal{M}$ is in the state q_i, the module $\mathcal{M}_i$ will be initialised (via the function z_i). In turn, $\mathcal{M}_i$ will feed the machine $\mathcal{M}$ with appropriate inputs (via the function y_i). These inputs will be processed by the ϕ's that label arcs emerging from q_i in the control machine $\mathcal{M}$.

The refinement we shall be defining will be a system consisting of the control machine $\mathcal{M}$ and the modules $\mathcal{M}_i$. The way in which these communicate can be seen as a process in which the modules $\mathcal{M}_i$ receive inputs from the external environment and feed $\mathcal{M}$ with appropriate inputs (see figure 1). Only one $\mathcal{M}_i$ is active at a time, depending on the state $\mathcal{M}$ is in. The whole process works as follows.

Let us assume that $\mathcal{M}_i$ is active and let

$$(q_i, m) \in Q \times M \text{ and } (p_i, m_i) \in (P_i - \{p_{if}\}) \times M_i,$$

be the current states and memory values of $\mathcal{M}$ and $\mathcal{M}_i$ respectively. Also, let $\sigma \in \Sigma$ be the input received by $\mathcal{M}_i$. Then we have the following situations:

i) σ causes $\mathcal{M}_i$ to halt (i.e. there is no transition on σ from p_i and m_i). In this case the whole system will halt.

ii) σ causes $\mathcal{M}_i$ to go to a new state $p_i' \neq p_{if}$. Then $\mathcal{M}_i$ will remain active and the state and memory values of $\mathcal{M}$ will remain unchanged. In this case, the output produced by the whole system will be the output $\gamma_1 \in \Gamma_1$ produced by $\mathcal{M}_i$. In this case we say that the whole system performs a *type A* transition (see figure 2).

iii) σ causes $\mathcal{M}_i$ to go to p_{if}; let m_i' be the new memory value of $\mathcal{M}_i$. Then $\mathcal{M}_i$ becomes inactive and $\mathcal{M}$ receives the input $y_i(m_i')$. If this input causes $\mathcal{M}$ to halt, then the whole system will halt. Otherwise, let q_j and m' be the new state and memory value of $\mathcal{M}$

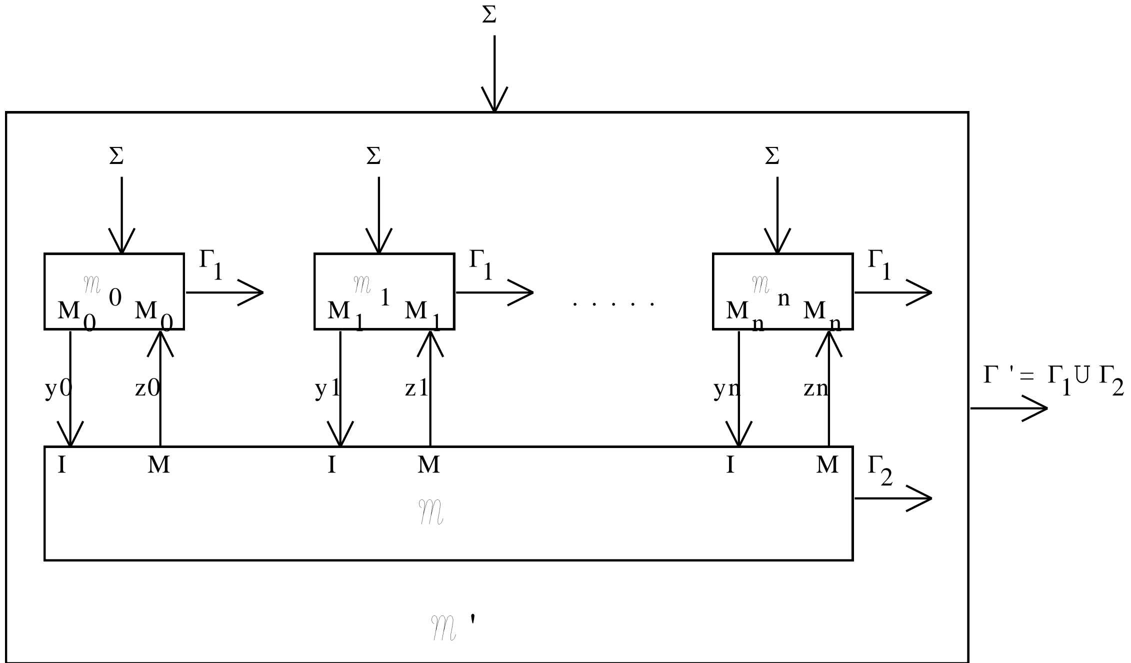

Figure 1.

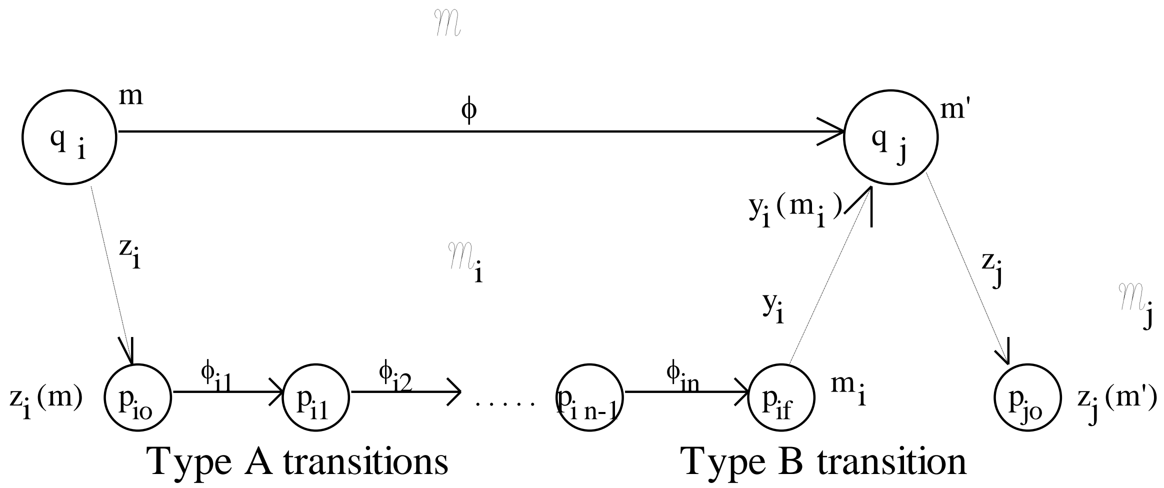

Figure 2.

respectively (i.e. the input $y_i(m_i')$ takes $\mathcal{M}$ from q_i and m to q_j and m'). Then the module $\mathcal{M}_j$ will become active; the current state of $\mathcal{M}_j$ will be p_{jo} (i.e. the initial state of the module $\mathcal{M}_j$)

and the current memory value $z_j(m')$. The output produced by the whole system will be γ_2, where $\gamma_2 \in \Gamma_2$ is the output produced by $\mathcal{M}$ when $y_i(m_i')$ is received in q_i and m. In this case we say that the whole system performs a *type B* transition (see figure 2).

More formally, we have the following definition.

Definition 2.2.

Let $\mathcal{M}_i = (\Sigma, \Gamma_1, P_i, M_i, \Phi_i, F_i, p_{io})$ be deterministic stream X-modules as above and let $z_i: M \to M_i$, $y_i: M_i \to I$ be functions, i = 0, .., n. Also, let $\mathcal{M} = (I, \Gamma_2, Q, M, \Phi, F, q_o, m_o)$ be a deterministic stream X-machine with $Q = \{q_o, ..., q_n\}$ and let *Ref*: $Q \to \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a function defined by $Ref(q_i) = (z_i, y_i, \mathcal{M}_i)$, i = 0, ..., n. We define a stream X-machine

$$\mathcal{M}' = (\Sigma, \Gamma', Q', M', q_o', m_o', \Phi', F')$$

as follows.

1. The input alphabet is Σ.
2. The output alphabet is $\Gamma' = \Gamma_1 \cup \Gamma_2$.
3. The state set is $Q' = \bigcup_{i=0}^{n} \{q_i\} \times (P_i - \{p_{if}\})$.

4. The memory is $M' = M \times M_a$, where $M_a = \bigcup_{i=0}^{n} M_i$.

5. The initial state is $q_o' = (q_o, p_{oo})$.

6. The initial memory value is $m_o' = (m_o, z_o(m_o))$.

7. The basic functions Φ' are derived from the basic functions Φ and Φ_i by application of two constructed functions *c* and *d*, which are defined below. Then

$\Phi' = \Phi_1' \cup \Phi_2'$, where

$\Phi_1' = \{c(\phi_i) | \phi_i \in \Phi_i, i = 0, ..., n\}$ and

$\Phi_2' = \{d(q_i, \phi, \phi_i) | \phi \in \Phi, \phi_i \in \Phi_i$ such that $F(q_i, \phi) \neq \emptyset, i = 0,.., n\}$.

i) Let $i \in \{0, ..., n\}$ and $\phi_i \in \Phi_i$. Then

$c(\phi_i): M' \times \Sigma \to \Gamma' \times M'$

is a (partial) function defined by:

$$c(\phi_i)((m, m_a), \sigma) = \begin{cases} (\gamma_1, (m, m_a')), \text{ if } m_a \in M_i \text{ and } (m_a, \sigma) \in \text{dom } \phi_i \\ \emptyset, \text{ otherwise} \end{cases}$$

$\forall \sigma \in \Sigma, m \in M, m_a \in M_a$, where

$\gamma_1 \in \Gamma_1, m_a' \in M_a$ satisfy $(\gamma_1, m_a') = \phi_i(m_a, \sigma)$.

So, Φ_1' are the functions Φ suitably embedded as functions acting on $M' \times \Sigma$. These correspond to the type A transitions above.

ii) Let $i \in \{0, ..., n\}, \phi \in \Phi, \phi_i \in \Phi_i$, such that

$F(q_i, \phi) \neq \emptyset$ (i.e. there is an arc labelled ϕ in $\mathcal{M}$ that emerges from q_i) and

$\exists\ p_i \in P_i$ such that $F_i(p_i, \phi_i) = p_{if}$ (i.e. there exists an arc labelled ϕ_i in $\mathcal{M}_i$ that enters p_{if}). Then

$$d(q_i, \phi, \phi_i): M' \times \Sigma \rightarrow \Gamma' \times M'$$

is a (partial) function defined as follows.

Let $\sigma \in \Sigma$, $m \in M$, $m_a \in M_a$. If $m_a \in M_i$ and $(m_a, \sigma) \in \text{dom } \phi_i$, then let $m_i' \in M_i$ such that

$$\phi_i(m_a, \sigma) = (\delta, m_i'), \text{ where } \delta \in \Gamma_1 \text{ is the end marker.}$$

Obviously, $y_i(m_i') \in I$. Also, if $(m, y_i(m_i')) \in \text{dom } \phi$, then let $\gamma_2 \in \Gamma_2$, $m' \in M$ such that

$$\phi(m, y_i(m_i')) = (\gamma_2, m').$$

Then

$$d(q_i, \phi, \phi_i)((m, m_a), \sigma) = \begin{cases} (\gamma_2, (m', m_a')), \text{ if } m_a \in M_i, (m_a, \sigma) \in \text{dom } \phi_i \\ \qquad\qquad\qquad \text{and } (m, y_i(m_i')) \in \text{dom } \phi \\ \varnothing, \text{ otherwise} \end{cases}$$

$\forall\ \sigma \in \Sigma$, $m \in M$, $m_a \in M_a$, where

$\gamma_2 \in \Gamma_2$, $m' \in M$, $m_i' \in M_i$ are defined as above and

$m_a' = z_j(m')$,

where j is chosen such that $q_j = F(q_i, \phi)$.

So, if $q_i \xrightarrow{\phi} q_j$ is an arc in $\mathcal{M}$ and $\phi_i \in \Phi_i$, then $\phi' = d(q_i, \phi, \phi_i)$ is obtained by applying ϕ_i and ϕ one after the other. The input received by ϕ will be the next memory value computed by ϕ_i transformed through the function y_i. The processing functions $d(q_i, \phi, \phi_i)$ will correspond to type B transitions above. The module currently 'active' ($\mathcal{M}_i$) is

'deactivated' and a new module ($\mathcal{M}_j$) is 'activated'. The function z_j is used to initialise the module $\mathcal{M}_j$.

8. The next state function F' is defined by:

$$F'((q_i, p_i), \phi') = \begin{cases} (q_i, F_i(p_i, \phi_i)), \text{ if } \exists\, \phi_i \in \Phi_i \text{ such that } c(\phi_i) = \phi' \\ \qquad\qquad \text{and } F_i(p_i, \phi_i) \neq p_{if} \\ (q_j, p_{jo}), \text{ if } \exists\, \phi \in \Phi, \phi_i \in \Phi_i \text{ such that } d(q_i, \phi, \phi_i) = \phi' \\ \qquad\qquad \text{and } F_i(p_i, \phi_i) = p_{if} \\ \qquad\qquad \text{where j is chosen such that } F(q_i, \phi) = q_j, \\ \varnothing, \text{ otherwise} \end{cases}$$

$\forall\, i \in \{0, ..., n\}$, $p_i \in P_i - \{p_{if}\}$ and $\phi' \in \Phi'$.

So, if $p_i \xrightarrow{\phi_i} p_i'$ is an arc in $\mathcal{M}_i$ and $p_i' \neq p_{if}$ then

$$(q_i, p_i) \xrightarrow{c(\phi_i)} (q_i, p_i')$$

is an arc in $\mathcal{M}'$. This corresponds to type A transitions.

If $p_i \xrightarrow{\phi_i} p_{if}$ is an arc in $\mathcal{M}_i$ and $q_i \xrightarrow{\phi} q_j$ is an arc in $\mathcal{M}$, then

$$(q_i, p_i) \xrightarrow{d(q_i, \phi, \phi_i)} (q_j, p_{jo})$$

is an arc in $\mathcal{M}'$. This corresponds to type B transitions.

Then $\mathcal{M}'$ is said to be a *refinement* of $\mathcal{M}$ *w.r.t.* Ref. The function Ref is called the *refinement function*. We also say that $(z_i, y_i, \mathcal{M}_i)$ *refines* q_i (written $(z_i, y_i, \mathcal{M}_i) = ref(q_i)$). The set $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ is called the *refinement set*. The functions y_i and z_i will be called the *input transfer functions* and *memory transfer functions* respectively.

It can be proven easily (see Ipate [5]) that F' is well defined and that $\mathcal{M}'$ is deterministic.

There may be some states in the machine $\mathcal{M}$ in which the arcs that emerge from them need not be refined (i.e. in the state q_i, $\mathcal{M}$ reads its inputs directly from the external environment). In this case the module $\mathcal{M}_i$ attached to the state q_i will have the form $\mathcal{M}_i = (\Sigma, \Gamma_1, P_i, M_i, \Phi_i, F_i, p_{io})$, where:

1. The state set is $\{p_{io}, p_{if}\}$.
2. The output alphabet is $\Gamma = \{\delta\}$.
3. The memory is $M_i = \Sigma$.
4. The type is $\Phi_i = \{\phi_i\}$, with $\phi_i: \Sigma \times \Sigma \rightarrow \{\delta\} \times \Sigma$ a function defined by:

   $\phi_i(\sigma, \sigma') = (\delta, \sigma')\ \forall\ \sigma, \sigma' \in \Sigma.$

5. The state transition diagram determined by F_i consists of a single arc from p_{io} to p_{if} labelled ϕ_i, i.e. F_i is a partial function defined by dom $F_i = \{(p_{io}, \phi_i)\}$ and $F_i(p_{io}, \phi_i) = p_{if}$.

In other words, $\mathcal{M}_i$ is used only to store the new input read and does nothing apart from this. We call this a *trivial* refinement module.

In this case the definition of memory transfer function $z_i: M \rightarrow \Sigma$ is irrelevant (i.e. it does not affect the construction of the refined machine). For example, it can be chosen to be a constant function. The input transfer function $y_i: \Sigma \rightarrow I$ will be an injective function that converts an input into the corresponding input *in* $\in$ I for the control machine; the actual

expression of y_i will depend on how I is chosen. For example, if $\Sigma \subseteq I$, then y_i will be the identity function.

The refinement described above allows machine specifications to be developed gradually. Instead of constructing the whole specification in a single step, a skeleton of the system (the 'control' machine $\mathcal{M}$) is produced first. This will use some fictitious inputs I. The way in which these inputs are obtained from the real ones (those that come from outside) is then specified by the sub-modules $\mathcal{M}_i$. The transfer functions z_i and y_i are usually very simple (identities, projections, constant functions, etc.).

The method is advantageous when we are dealing with complex systems. For example, it can be used to separate the user interface from the core functionality of the system (a simple example will be given later on, see section 3). Furthermore, by providing a coarse specification first and showing explicitly how this is refined, we ease the understanding of complex specifications. A specification consisting of many simple machines linked in a well defined way could give us a better idea of what the system is supposed to do than a single, but very complex machine.

3. Refinement - a simple example.

We present a stream X-machine specification of a simple program which enables users to log in to a computer system using their username and password and to replace their password. We require that each user can enter his/her password only once. The inputs

used are: character keys, the Enter key and the Backspace key. Only usernames and of at most n characters and password of at most m characters will be considered valid. Thus, if more than n (or m) characters have been entered, the rest will be ignored, unless one or more of the first n (or m) characters have been deleted. The system will display the valid characters (i.e. less then n) of the username, but nothing will be displayed when the password is entered.

3.1. The control machine.

We shall construct the stream X-machine specification $\mathcal{M}'$ of the program in two stages. First, we give an 'unrefined' stream X-machine specification $\mathcal{M}$. This will operate on sequences of characters. The way in which these sequences of characters are entered and fed to $\mathcal{M}$ will be detailed using the operation of refinement (i.e. $\mathcal{M}$ will be the control machine of the refinement). In what follows we shall assume that $m \geq n$.

The stream X-machine $\mathcal{M}$ is defined as follows.

1. The input set is $I = STRINGS_m$,

where

$STRINGS_m = \sum_{k=o}^{m}$ $CHARACTERS^k$ is the set of sequences of at most m characters.

2. The output is Γ_2 = MESSAGES, where

MESSAGES = {*msg1*, ..., *msg5*}) is a set of messages given by the program (e.g. *msg1* = 'type in your password', etc.).

3. The set of states is:

$Q = \{q_o, q_1, q_2\}$. The initial state is q_o.

4. The memory is M = ACCOUNT_INFO × $STRINGS_n$, where

ACCOUNT_INFO = $STRINGS_n \rightarrow STRINGS_m$ is the set of partial functions from $STRINGS_n$ to $STRINGS_m$.

A memory value will be a tuple (*acc*, *name*),, where *acc* ∈ACCOUNT_INFO will keep the mapping between the usernames and password for the existing accounts and *name* ∈ $STRINGS_n$ will store the username entered by the current user.

5. The initial memory value is

m_o = (*in_acc*, *in_name*).

6. The type is:

Φ = {good_name, wrong_name, good_psw, wrong_psw, new_psw}.

7. The state transition diagram is represented in figure 3.

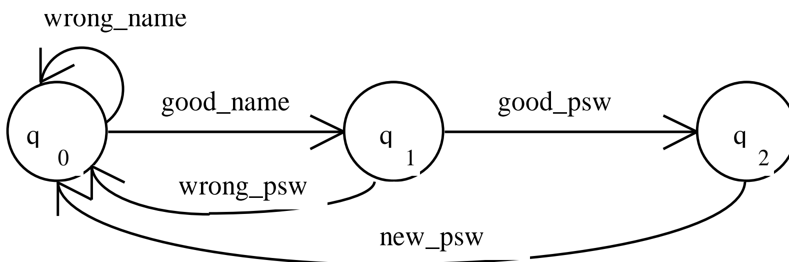

Figure 3.

8. The basic processing functions are (partial) functions from $I \times M$ to $\Gamma_2 \times M$. In what follows we define them in terms of their domains and expressions.

· dom `good_name` = $\{((acc, name), str) \in M \times STRINGS_n \mid str \in \text{dom } acc\}$

`good_name`$((acc, name), str) = (msg1, (acc, str))$

· dom `wrong_name` = $\{((acc, name), str) \in M \times STRINGS_n \mid \neg(str \in \text{dom } acc)\}$

`wrong_name`$((acc, name), str) = (msg2, (acc, name))$

· dom `good_psw` = $\{((acc, name), str) \in M \times STRINGS_m \mid$

$name \in \text{dom } acc \text{ and } acc(name) = str\}$

`good_psw`$((acc, name), str) = (msg3, (acc, name))$

· dom `wrong_psw` = $\{(acc, name), str) \in M \times STRINGS_m \mid$

$\neg(name \in \text{dom } acc \text{ and } acc(name) = str)\}$

`wrong_psw`$((acc, name), str) = (msg4, (acc, name))$

· dom `new_psw` = $M \times STRINGS_m$

`new_psw`$((acc, name), str) = (msg5, (acc', name))$

where $acc' = acc \oplus (name, str)$

3.2. The refinement function.

The stream X-machine $\mathcal{M}$ is refined using the refinement function

$$\text{Ref}: \{q_i\}_{i = 0,...2} \rightarrow \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...2} \text{ defined by:}$$

$$\text{Ref}(q_i) = (z_i, y_i, \mathcal{M}_i), i = 0, ..., 2.$$

3.3. The refinement modules.

$\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$ are stream X-modules with the input and output alphabets Σ and Γ_1 respectively, where:

Σ = CHARACTERS $\cup$ {`back_space`, `enter`}

(i.e. Σ contains all the keys allowed),

Γ_1 = DISPLAYS $\cup$ {`delete_char`, `empty_display`} $\cup$ {δ},

where DISPLAYS is the set of displays of all characters, i.e. there is a bijective function `display`: CHARACTERS $\rightarrow$ DISPLAYS.

Then, the three modules are defined as follows.

A. $\mathcal{M}_0$.

1. The state set is $P_0 = \{p_{0.0}, ..., p_{0.n}, p_{0.n+1}\}$; $p_{0.0}$ is the initial state; $p_{0.n+1}$ is the final state.

2. M_0 = STRINGS_n. We denote by *str* $\in$ STRINGS_n the current memory value.

3. The type is:

$\Phi_0 = \{$type_ch1, type_ch2, press_bs1, press_bs2, press_enter1$\}$

4. The transition diagram is shown in figure 4.

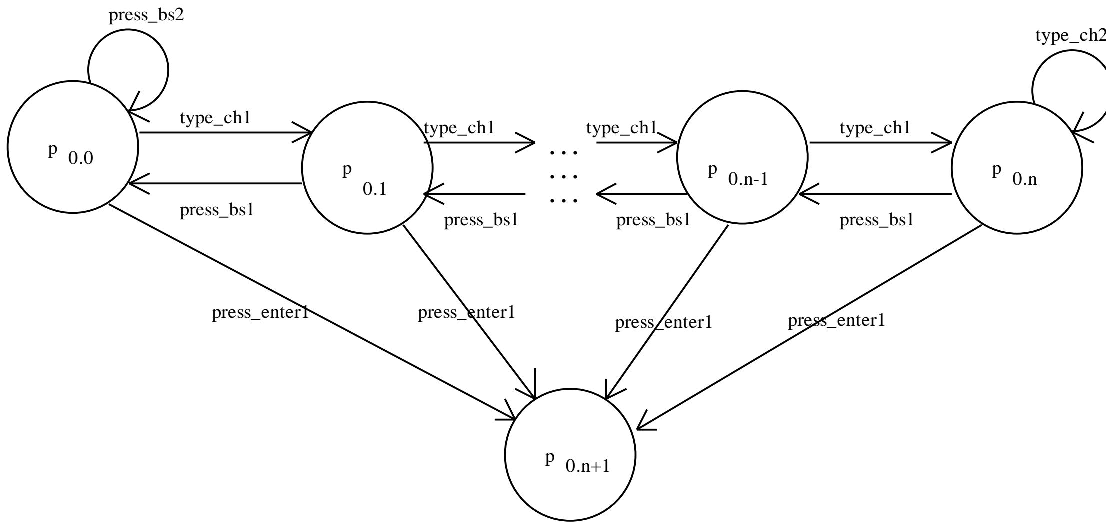

Figure 4.

5. The basic functions are defined by:

· dom type_ch1 = $\text{STRINGS}_n \times \text{CHARACTERS}$

type_ch1(str, ch) = (display(ch), str :: ch)

i.e. the character ch is displayed and added at the end of $str \in \text{STRINGS}_n$.

· dom type_ch2 = $\text{STRINGS}_n \times \text{CHARACTERS}$

type_ch2(str, ch) = ($empty_display$, str)

· dom press_bs1 = $\text{STRINGS}_n \times \{back_space\}$

press_bs1(*str*, *back_space*) = (*delete_char*, tail(*str*))

i.e. the last character of the sequence *str* is removed.

· dom press_bs2 = $STRINGS_n$ × {*back_space*}

press_bs2(*str*, *back_space*) = (*empty_display*, *str*)

· dom press_enter1 = $STRINGS_n$ × {*enter*}

press_enter1(*str*, *enter*) = (δ, *str*)

B. $\mathcal{M}_1$.

1. The state set is $P_1 = \{p_{1.0}, ..., p_{1.m}, p_{1.m+1}\}$; $p_{1.0}$ is the initial state; $p_{1.m+1}$ is the final state.

2. $M_1 = STRINGS_n$. We denote by *str* ∈ $STRINGS_m$ the current memory value.

3. The type is:

Φ_1 = {type_ch3, type_ch4, press_bs3, press_bs4, press_enter2}

4. The transition diagram is shown in figure 5.

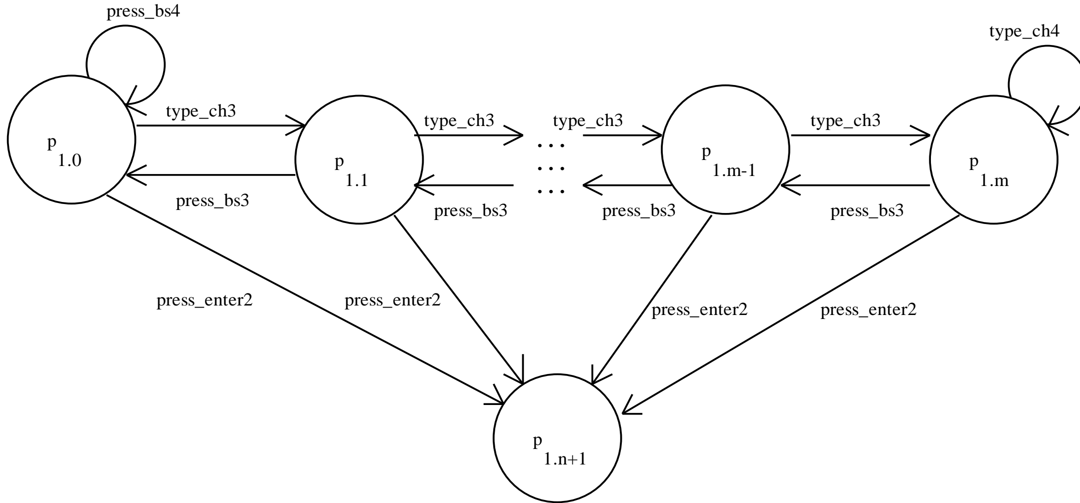

Figure 5.

5. The basic functions are defined by:

· dom `type_ch3` = STRINGS_m × CHARACTERS

`type_ch3`(*str*, *ch*) = (*`empty_display`*, *str* :: *ch*)

· dom `type_ch4` = STRINGS_m × CHARACTERS

`type_ch4`(*str*, *ch*) = (*`empty_display`*, *str*)

· dom `press_bs3` = STRINGS_m × {*`back_space`*}

`press_bs3`(*str*, *`back_space`*) = (*`delete_char`*, tail(*str*))

· dom `press_bs4` = STRINGS_m × {*`back_space`*}

`press_bs2`(*str*, *`back_space`*) = (*`empty_display`*, *str*)

· dom `press_enter2` = $STRINGS_m \times$ {*enter*}

`press_enter2`(*str*, *enter*) = (δ, *str*)

C. $\mathcal{M}_2$ is identical to $\mathcal{M}_1$.

3.4. The transfer functions.

z_0: M → $STRINGS_n$ and y_0:$STRINGS_n$ → $STRINGS_m$ are defined by:

$z_0(m) = \varepsilon, \forall m \in M$, where ε denotes the empty sequence.

$y_0(str) = str, \forall str \in STRINGS_n$ (i.e. we are assuming that $m \geq n$).

For i = 1, 2, z_i: M → $STRINGS_m$ and y_i: $STRINGS_m$ → $STRINGS_m$ are defined by:

$z_i(m) = \varepsilon, \forall m \in M$,

$y_i(str) = str, \forall str \in STRINGS_m$.

3.5. The refined machine.

The specification of the system is the stream X-machine $\mathcal{M}'$ which is the refinement of $\mathcal{M}$ w.r.t. Ref. Then $\mathcal{M}'$ is defined as follows:

1. The input set is

   Σ = CHARACTERS ∪ {*back_space*, *enter*}.

2. The output set is $\Gamma' = \Gamma_1 \cup \Gamma_2 - \{\delta\}$ (i.e. obviously, δ can be omitted since it will not appear in the refined machine). Therefore

Γ' = DISPLAYS $\cup$ {*delete_char*, *empty_display*} $\cup$ MESSAGES.

3. The state set is

$Q' = \{q_{0.0}, ..., q_{0.n}, q_{1.0}, ..., q_{1.m}, q_{2.0}, ..., q_{2.m}\}$

(i.e. $q_{0.0} = (q_0, p_{0.0})$, ..., $q_{0.n} = (q_0, p_{0.n})$, $q_{1.0} = (q_1, p_{1.0})$, ..., $q_{1.m} = (q_1, p_{1.m})$,

$q_{2.0} = (q_2, p_{1.0})$, ..., $q_{2.m} = (q_1, p_{1.m})$)

4. The memory is

M' = ACCOUNT_INFO × STRINGS_n × STRINGS_m

(i.e. M' = M × M_a, where M = ACCOUNT_INFO × STRINGS_n and M_a = STRINGS_m).

The initial memory value is

m_o' = (*in_acc*, *in_name*, ε).

5. The type is

Φ' = {type_ch1', type_ch2', type_ch3', type_ch4',

press_bs1', press_bs2', press_bs3', press_bs4',

press_enter1.1', press_enter1.2', press_enter2.1',

press_enter2.2', press_enter2.3'}, where

type_ch1' = *c*(type_ch1), ..., type_ch4' = *c*(type_ch4), ...

press_bs1' = *c*(press_bs1), ..., press_bs4' = *c*(press_bs4)

press_enter1.1' = *d*(q_o, wrong_name, press_enter1),

press_enter1.2' = *d*(q_o, good_name, press_enter1),

press_enter2.1' = d(q$_1$, wrong_psw, press_enter2),

press_enter2.2' = d(q$_1$, good_psw, press_enter2),

press_enter2.3' = d(q$_2$, new_psw, press_enter2)

(i.e. the other functions obtained by applying c and d will not appear in the transition diagram of $\mathcal{M}$', so we ignore them).

For example type_ch1', press_enter1.2', press_enter2.1' are defined by:

· dom type_ch1' = (ACCOUNT_INFO × STRINGS$_n$ × STRINGS$_n$) × CHARACTERS

type_ch1'((acc, $name$, str), ch) = (display(ch), (acc, $name$, str :: ch))

· dom press_enter1.2' = {((acc, $name$, str), *enter*)| $acc \in$ ACCOUNT_INFO,

$name$, $str \in$ STRINGS$_n$ such that $str \in$ dom acc }

press_enter1.2'((acc, $name$, str), *enter*) = (*msg1*, (acc, str, ε)) .

6. The state transition diagram is represented in figure 6.

Figure 6.

4. Implementation.

A specification developed using the refinement approach admits a straightforward implementation. The refined machine need not be constructed explicitly. Instead, the whole system can be implemented as a 'master' implementation (program) $\mathcal{I}$ that calls the subimplementations (subroutines) $\mathcal{I}_i$. $\mathcal{I}$ is obtained from the implementation of the control machine $\mathcal{M}$ by replacing instructions that read inputs from the outside environment with instructions that call the subroutines $\mathcal{I}_i$. These subroutines will feed $\mathcal{I}$ with appropriate inputs. Basically, the execution of $\mathcal{I}$ will call a subroutine $\mathcal{I}_i$ before executing any $\phi \in \Phi$.

A subimplementation $\mathcal{I}_i$ will have the form:

apply z_i (initialise the subroutine);

implement $\mathcal{M}_i$; the final state p_{if} will correspond to the 'exit' state of this subroutine;

apply y_i (return a variable or a set of variables $v \in I$ that will be used as inputs in $\mathcal{I}$).

In this case, we shall say that $\mathcal{I}_i$ is the *implementation* of $(z_i, y_i, \mathcal{M}_i)$.

Since z_i and y_i are usually very simple functions, writing the program $\mathcal{I}_i$ will consist mainly of implementing the X-module $\mathcal{M}_i$.

If a module $\mathcal{M}_i$ is trivial, then $\mathcal{I}_i$ will just store the input received and convert it (if necessary) into a suitable input for the control machine $\mathcal{M}$.

5. Testing.

Let $\mathcal{M}'$ be a specification of a system constructed using the refinement operation described above. Then, one way of testing the implementation of the system is to construct $\mathcal{M}'$ explicitly, implement it and test the implementation against $\mathcal{M}'$ using the method presented in Ipate and Holcombe [6]. This approach might not be always convenient for the following reasons. Firstly, it requires $\mathcal{M}'$ to be constructed explicitly, which might not be desirable if we are dealing with large systems. In this case it might be more convenient to implement the refinement modules separately and to construct the implementation of the system as described in the previous section. Secondly, if the state set is very large, then the test set obtained will also be large.

An alternative to this is to use a two phase approach in which the basic modules are implemented and tested separately and the whole system is tested for integration. Since the modules $\mathcal{M}_i$ can be tested using the method presented in Ipate and Holcombe [6], we shall discuss now how the system integration can be tested.

As in Ipate and Holcombe [6], we shall be using a reductionist approach. We shall assume that the implementation of the system is constructed from the following components:

· the *correct* implementations of the basic functions Φ of the control machine $\mathcal{M}$;

· the subimplementations $\mathcal{I}_i$ ($\mathcal{I}_i$ is the implementation of (z_i, y_i, $\mathcal{M}_i$) as described in the previous section). These can be separate subprograms or pieces of code that can be identified in the main implementation. We assume that the implementation of the transfer functions z_i and y_i are correct (these are usually simple functions). We also assume that the implementations of the refinement modules have been tested against their specifications.

If these conditions are met, then the implementation of the whole system can be viewed as a control program $\mathcal{I}$, that receives inputs from the subprograms $\mathcal{I}_i$. Since the basic functions Φ are assumed to be correct, $\mathcal{I}$ will be the implementation of a stream X-machine $\mathcal{M}^*$ with the same type Φ (of course in this implementation instructions that read inputs from the outside environment are replaced by instructions that call the subimplementations $\mathcal{I}_i$)

Now, let q^* be a state in $\mathcal{M}^*$. Then, there will be a subimplementation $\mathcal{I}_i$ such that $\mathcal{I}$ receives inputs from $\mathcal{I}_i$ when $\mathcal{M}^*$ is in the state q^*. Let $\mathcal{M}_i^*$ be the implementation of $\mathcal{M}_i$ and $\mathcal{A}_i$ and $\mathcal{A}_i^*$ the associated automata of $\mathcal{M}_i$ and $\mathcal{M}_i^*$ respectively. If $\mathcal{M}_i^*$ has been tested against $\mathcal{M}_i$ using the stream X-machine testing method, then $\mathcal{A}_i$ and Min($\mathcal{A}_i^*$) are isomorphic, where Min($\mathcal{A}_i^*$) is the minimal automaton of $\mathcal{A}_i^*$.

Then the whole system behaves as though $(z_i, y_i, \mathcal{M}_i)$ *refines* the state q*. Indeed, the fact that $\mathcal{A}_i$* might not be minimal does not make any difference as far as the main program $\mathcal{I}$ is concerned (i.e. two equivalent states will behave identically as far as $\mathcal{I}$ is concerned).

Therefore, the implementation of the whole system (let us call this $\mathcal{M}$'*) can be modelled as the refinement of $\mathcal{M}$* w.r.t. Ref*, where Ref* is a refinement function Ref*: Q* → $\{(z_i, y_i, \mathcal{M}_i)\}_{i=0,...,n}$, where Q* is the state set of $\mathcal{M}$* and $\{(z_i, y_i, \mathcal{M}_i)\}_{i=0,...,n}$ is the refinement set of the specification (see figure 7).

Thus, once the basic functions Φ and the refinement modules $\mathcal{M}_i$ have been tested, the integration testing will be carried out to ensure that:

i) the implementation of the control machine $\mathcal{M}$ is correct (therefore testing $\mathcal{M}$* against $\mathcal{M}$)

ii) each state in $\mathcal{M}$* is refined by the appropriate module $\mathcal{M}_i$ (therefore testing Ref* against the refinement function of the specification, Ref).

Therefore, our testing method constructs a test set that tests the system for integration in the sense mentioned above, provided that the assumptions we have made at the beginning of the section are met.

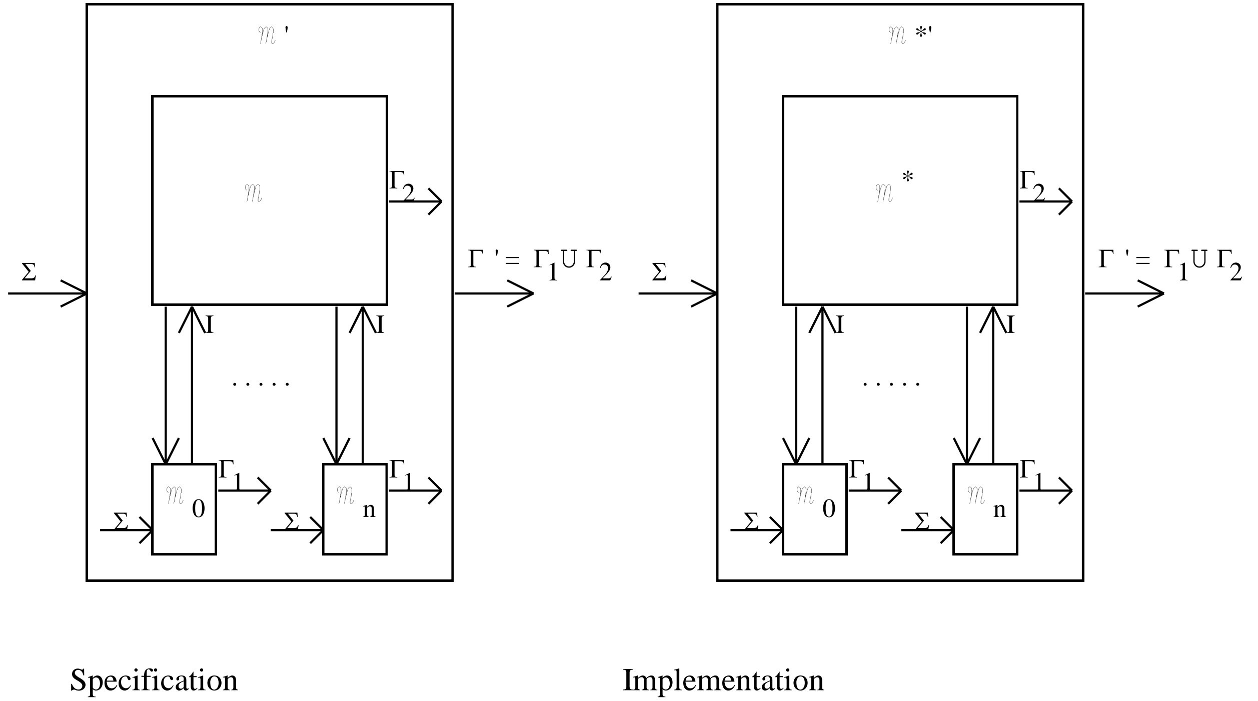

Figure 7.

5.1. 'Design for testing' conditions.

Similar to the testing method presented in Ipate and Holcombe [6], our specification has to satisfy some 'design for testing' conditions.

5.1.1. Simple refinement.

Notation 5.1.1.1

Let $\mathcal{M}_i$ and $\mathcal{M}_j$ be two stream X-modules and $\mathcal{A}_i$ and $\mathcal{A}_j$ their associated automata. Then $\mathcal{M}_i \equiv \mathcal{M}_j$ denotes that $\mathcal{M}_i$ and $\mathcal{M}_j$ have the same type ($\Phi_i = \Phi_j$) and $\mathcal{A}_i$ and $\mathcal{A}_j$ are isomorphic.

Definition 5.1.1.2.

Let $\mathcal{M}'$ be the refinement of a stream X-machine $\mathcal{M}$ w.r.t. Ref, where Ref: $Q \rightarrow \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,..,n}$ is the refinement function. Then we say that $\mathcal{M}'$ is a *simple refinement* if $\forall$ i, j $\in \{0, ..., n\}$ $\mathcal{M}_i \equiv \mathcal{M}_j$ implies $y_i = y_j$ and $z_i = z_j$.

For instance, the refinement given in section 3 is simple.

5.1.2. Complete refinement.

For similar reasons to those given in Ipate and Holcombe [6], our testing method requires some completeness and output-distinguishability conditions.

Definition 5.1.2.1.

Let $\mathcal{M}'$ be the refinement of a stream X-machine $\mathcal{M}$ w.r.t. Ref, Ref: $Q \rightarrow \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,..,n}$. Let i $\in \{0, ..., n\}$ and let u_{ie} be the generalised transition function of $\mathcal{M}_i$ (see Ipate and Holcombe [6]). Then we say that $\mathcal{M}_i$ is *complete w.r.t.* $\phi \in \Phi$ if:

$\forall m \in M, \exists s \in \Sigma^*$ with $|s| \geq 2$ and $m_i \in M_i$ such that

$$u_{ie}(p_{io}, z_i(m), s) = (p_{if}, m_i)$$

and

$$(m, y_i(m_i)) \in \text{dom } \phi.$$

In other words s is a string of length at least two that takes the module $\mathcal{M}_i$ from the initial state p_{io} and memory value $z_i(m)$ to the final state p_{if} and m_i such that $(m, y_i(m_i)) \in \mathrm{dom}\ \phi$.

Definition 5.1.2.2.

Let $\mathcal{M}'$ be the refinement of a stream X-machine $\mathcal{M}$ w.r.t. Ref with Ref: $Q \to \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,..,n}$. Then $\mathcal{M}'$ is said to be a *complete refinement* if:

$\forall$ i $\in$ {0, ..., n} and $\phi \in \Phi$, $\mathcal{M}_i$ is complete w.r.t. ϕ.

Example 5.1.2.3.

It can be shown very easily that the actual memory (the set of the memory values that can be actually taken by the machine for all possible computations) of the control machine described in section 3 is

M_r = {(*acc*, *name*) | (dom *acc* = dom `in_acc`) and

((*name* $\in$ dom `in_acc`) or (*name* = `in_name`))}

If we assume that `in_name` $\in$ dom `in_acc` (i.e. `in_name` is a valid username), m = n and that `in_acc` is not a total function (i.e. not all strings of length at most n are valid usernames) it can be verified that $\forall$ i $\in$ {0, 1, 2} and $\phi \in \Phi$, $\mathcal{M}_i$ is complete w.r.t. ϕ (i.e. the completeness is considered with respect to the actual memory M_r).

The completeness condition can be enforced by simply extending the definitions of the processing functions of the control machine and adding two new symbols to the input

alphabet Σ (see Ipate[5]). This augmentation corresponds to adding some extra code to the implementation in a very straightforward manner. This code can be easily removed after testing has been completed.

5.1.3. Output-distinguishable refinement set.

The second condition required by our refinement testing method will be the output-distinguishability of the modules $\mathcal{M}_i$. Let us define the set

$$\Phi_i[p_{io}] = \{\phi_i \in \Phi_i | F_i(p_{io}, \phi_i) \neq \varnothing \text{ and } F_i(p_{io}, \phi_i) \neq p_{if}\}$$

(in other words the set $\Phi_i[p_{io}]$ contains all ϕ_i's that are labels of arcs emerging from the initial state of the module $\mathcal{M}_i$ but not leading to its final state.

Then we have the following definitions.

Definition 5.1.3.1.

Let $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a refinement set. Let i, j $\in \{0,..., n\}$, $\phi_i \in \Phi_i$ and $\phi_j \in \Phi_j$. Then ϕ_i and ϕ_j are said to be *fully output-distinguishable* if:

$\forall\, \sigma \in \Sigma$, $m_i \in M_i$, $m_j \in M_j$, if $\phi_i(m_i, \sigma) = (m_i', \gamma)$, $\phi_j(m_j, \sigma) = (m_j', \gamma')$ with $m_i' \in M_i$, $m_j' \in M_j$, $\gamma, \gamma' \in \Gamma_1$, then $\gamma \neq \gamma'$.

In other words ϕ_i and ϕ_j produce different outputs on any input character, regardless of the memory values chosen.

Definition 5.1.3.2.

Let $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a refinement set and let i, j $\in$ {0, ..., n}. Then $\mathcal{M}_i$ and $\mathcal{M}_j$ are said to be *output-distinguishable* if $\forall$ $\phi_i \in \Phi_i[p_{io}]$ and $\phi_j \in \Phi_j[p_{jo}]$, ϕ_i and ϕ_j are fully output-distinguishable.

Definition 5.1.3.3.

Let $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a refinement set. Then $\mathcal{R}$ is said to be *output-distinguishable* if $\forall$ i, j $\in$ {0,.., n}, either $\mathcal{M}_i \equiv \mathcal{M}_j$ or $\mathcal{M}_i$ and $\mathcal{M}_j$ are output-distinguishable

This condition can be enforced by augmenting the output alphabet to $\Gamma_1 \times G_1$, where the number of elements of the extra component of the output, G_1, will be at most the number of non-identical (up to an isomorphism of the associated automata) refinement modules.

5.1.4. Fundamental test function and the refinement testing theorem.

As for stream X-machines (see Ipate and Holcombe [6]), we define a fundamental test function of a refinement that transforms sequences from Φ^* into sequences from Σ^*.

Definition 5.1.4.1.

Let $\mathcal{M}'$ be the refinement of a stream X-machine $\mathcal{M}$ w.r.t. Ref with Ref: $Q \rightarrow \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,..,n}$. We assume that $\mathcal{M}'$ is a complete refinement. Then we define recursively a function t: $\Phi^* \rightarrow \Sigma^*$ and a partial function t_M: $\Phi^* \rightarrow M$ as follows:

i).

- $t(\varepsilon) = \varepsilon$,
- $t_M(\varepsilon) = m_o$,

where m_o is the initial memory value of $\mathcal{M}$ and ε is the empty string.

ii). $\forall$ $k \geq 0$ and $\phi_1, ..., \phi_k, \phi_{k+1} \in \Phi$, the recursion step that defines $t(\phi_1... \phi_{k+1})$ and $t_M(\phi_1...\phi_k)$ function of $t(\phi_1... \phi_k)$ and $t_M(\phi_1...\phi_k)$ is as follows.

1). If there exists a path in $\mathcal{M}$ labelled $\phi_1... \phi_k$ that starts from q_o (i.e. q_o is the initial state of $\mathcal{M}$) then let q_i be the final state of this path.

Then:

- $t(\phi_1... \phi_{k+1}) = t(\phi_1... \phi_k)\ s_{k+1}$

where $s_{k+1} \in \Sigma^*$ with $|s_{k+1}| \geq 2$ is chosen such that

$\exists\ m_i \in M_i$ such that $u_{ie}(p_{io}, z_i(m), s_{k+1}) = (p_{if}, m_i)$ and $(m, y_i(m_i)) \in$ dom ϕ_{k+1}, where $m = t_M(\phi_1...\phi_k)$

Note: Since the refinement is complete, there exist such s_{k+1} and m_i.

The expression of $t_M(\phi_1...\phi_{k+1})$ depends on the following two cases.

1.a). If there exists an arc labelled ϕ_{k+1} that emerges from q_i, then

- $t_M(\phi_1...\phi_{k+1}) = m'$,

where m' is chosen such that $(m', \gamma) = \phi_{k+1}(m, y_i(m_i))$ for some $\gamma \in \Gamma_2$ (i.e. m' is the next memory value computed by ϕ_{k+1}), where $m_i \in M_i$ is the value from the above definition of $t(\phi_1 \ldots \phi_{k+1})$.

1.b) Otherwise

· $t_M(\phi_1 \ldots \phi_{k+1}) = \emptyset$.

2. If there is no path in $\mathcal{M}$ labelled $\phi_1 \ldots \phi_k$ that starts from q_o, then:

· $t(\phi_1 \ldots \phi_{k+1}) = t(\phi_1 \ldots \phi_k)$

· $t_M(\phi_1 \ldots \phi_{k+1}) = \emptyset$.

Then t is called a *fundamental test function* of the refinement.

The construction of the fundamental test function of a refinement is similar to that of a stream X-machine. Basically, if a path exists in the control machine $\mathcal{M}$, then the corresponding value of the test function will exercise that path. If a path does not exist in $\mathcal{M}$, then the corresponding value of the test function will exercise the part of the path that does exist in $\mathcal{M}$ plus one extra arc.

The reason for requiring $|s_{k+1}| \geq 2$ will become clear when we prove our refinement testing theorem. Basically, the first character of s_{k+1} (i.e. head(s_{k+1})) tests that a state in the implementation will be refined similarly to the corresponding state in the specification (i.e. the refinement module attached to those states are identical); the rest (i.e. tail(s_{k+1})) will be used to prove the equivalence of these states in the associated automata of the control machines.

Example 5.1.4.2.

Let $\mathcal{M}'$ be the refinement from section 3. We shall assume that *in_name* ∈ dom *in_acc*, m = n and that *in_acc* is not a total function. Hence, the refinement is complete. Let name1, name2 ∈ $STRINGS_n$ be a valid and an invalid user name respectively (i.e. name1 ∈ dom *in_acc*, name2 ∉ dom *in_acc*) let psw1, psw2, psw3 ∈ $STRINGS_n$ such that psw1 is a valid password for name1 (i.e. *in_acc*(name1) = psw1) and psw3 is a valid password for *in_name* (i.e. *in_acc*(*in_name*) = psw3). We also assume that |name1|, |name2|, |psw1|, |psw2|, |psw3| ≥ 1. Then, we illustrate the construction of t and t_M with the following examples.

t(good_name) = name1 *enter*

t_M(good_name) = (*in_acc*, name1)

t(good_name good_psw) = name1 *enter* psw1 *enter*

t_M(good_name good_psw) = (*in_acc*, name1)

t(good_name good_psw new_psw) = name1 *enter* psw1 *enter* psw2 *enter*

t_M(good_name good_psw new_psw) = (acc' name1)

where acc' = acc ⊕ (name1, psw2)

t(wrong_name) = name2 *enter*

t_M(wrong_name) = (*in_acc*, *in_name*)

t(wrong_name good_psw) = name2 *enter* psw3 *enter*

t_M(wrong_name good_psw) = ∅

t(wrong_name good_psw new_psw) = name2 *enter* psw3 *enter*

t_M(wrong_name good_psw new_psw) = ∅

We can now assemble the following result which is the basis for our refinement testing method. First, we make the following notation:

Definition 5.1.4.3.

Let Σ be an alphabet and $A \subseteq \Sigma^*$. Then the set *Pref(A)* denotes the set of all prefixes of A,

$$\text{Pref}(A) = \{s \in \Sigma^* \mid \exists\ a \in A, b \in \Sigma^* \text{ such that } sb = a\}.$$

Theorem 5.1.4.4.

Let $\mathcal{M} = (I, \Gamma_2, Q, M, \Phi, F, q_o, m_o)$, $\mathcal{M}^* = (I, \Gamma_2, Q^*, M, \Phi, F^*, q_o^*, m_o)$ be two stream X-machines with Φ output-distinguishable Let $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0, ..,n}$ be a refinement set, $\mathcal{M}_i = (\Sigma, \Gamma_1, P_i, M_i, \Phi_i, F_i, p_{io})$, z_i: $M \rightarrow M_i$, y_i: $M_i \rightarrow I$, $i = 0, ..., n$, and let Ref: $Q \rightarrow \mathcal{R}$ and Ref*: $Q^* \rightarrow \mathcal{R}$ be two refinement functions. We assume that $\mathcal{A}$, the associated automaton of $\mathcal{M}$, is minimal and that $\mathcal{R}$ is output-distinguishable. Let $\mathcal{M}'$ be the

refinement of $\mathcal{M}$ w.r.t. Ref and $\mathcal{M}^{*\prime}$ the refinement of $\mathcal{M}^*$ w.r.t. Ref* and let f and f* the functions computed by $\mathcal{M}'$ and $\mathcal{M}^{*\prime}$ respectively. We assume that $\mathcal{M}^{*\prime}$ is a simple and complete refinement. Let also T and W be a transition cover and a characterisation set of $\mathcal{A}$, $Z = \Phi^k W \cup \Phi^{k-1} W \cup ... \cup W$, with k a positive integer and $t: \Phi^* \rightarrow \Sigma^*$ a fundamental test function of the refinement $\mathcal{M}'$. If $card(Q^*) - card(Q) \leq k$, $\Gamma_1 \cap \Gamma_2 = \emptyset$ (i.e. the output alphabets of the control machine $\mathcal{M}$ and the refinement modules $\mathcal{M}_i$ are disjoint) and $f(s) = f^*(s), \forall\ s \in Pref(t(TZ))$, then $f = f^*$.

Note:

The definitions of transition cover and characterisation set of an automaton, output-distinguishable Φ, etc. are given in Ipate and Holcombe [6].

Proof:

First, let us make the following notation. Let

$$Q = \{q_o, q_1, ..., q_n\}$$

be the state set of $\mathcal{M}$ and let

$$Q^* = \{q_o^*, q_1^*, ..., q_{n'}^*\}$$

be the state set of $\mathcal{M}^*$ (q_o and q_o^* are the initial states). Without loss of generality we shall assume that the refinement function Ref: $Q \rightarrow \mathcal{R}$ is defined by

$$Ref(q_i) = (z_i, y_i, \mathcal{M}_i), i = 0, ..., n.$$

Also, for j = 0, ..., n', we shall denote

$$Ref(q_j^*) = (z_j^*, y_j^*, \mathcal{M}_j^*)$$

(of course $\forall$ j $\in$ {0, .., n'} $\exists$ i $\in$ {0, .., n} such that $z_j^* = z_i$, $y_j^* = y_i$, $\mathcal{M}_j^* = \mathcal{M}_i$).

Then we prove the following intermediary results.

Lemma 5.1.4.4.1.

1. Let $\phi_1, ..., \phi_k \in \Phi$ be such that there exists a path

$$q_0 \xrightarrow{\phi_1} q_{i_1} \xrightarrow{\phi_2} q_{i_2} \ldots q_{i_{k-1}} \xrightarrow{\phi_k} q_{i_k}$$

in $\mathcal{M}$ and let $t(\phi_1) = s_1$, $t(\phi_1...\phi_k) = s_1...s_k$.

If $f(s) = f^*(s)$, $\forall$ s $\in$ Pref($s_1...s_k$), then:

a. There exists a path $q_0^* \xrightarrow{\phi_1} q_{j_1}^* \xrightarrow{\phi_2} q_{j_2}^* \ldots q_{j_{k-1}}^* \xrightarrow{\phi_k} q_{i_k}^*$ in $\mathcal{M}^*$.

b. $z_o = z_o^*$, $y_o = y_o^*$, $\mathcal{M}_o \equiv \mathcal{M}_o^*$ and $z_{ir} = z_{jr}^*$, $y_{ir} = y_{jr}^*$, $\mathcal{M}_{ir} \equiv \mathcal{M}_{jr}^*$, r = 1, ..., k-1.

c. After receiving the string $s_1...s_r$, $r \leq k$, $\mathcal{M}'$ will be in the state (q_{ir}, p_{iro}) and $\mathcal{M}^{*\prime}$ will be in the state (q_{jr}^*, p_{jro}^*), where p_{iro} and p_{jro}^* are the initial states of $\mathcal{M}_{ir}$ and $\mathcal{M}_{jr}^*$ respectively. The corresponding memory values will be $(m_r, z_{ir}(m_r))$, $(m_r^*, z_{jr}(m_r^*))$, with $m_r, m_r^* \in M$.

d. $m_1 = m_1^*$, ..., $m_k = m_k^*$, where m_r and m_r^*, r = 1, ..., k, are the memory values from *c*.

2. Let $\phi_{k+1} \in \Phi$ such that there is no arc labelled ϕ_{k+1} emerging from q_{ik}. Let also $s_1...s_k s_{k+1} = t(\phi_1...\phi_k\phi_{k+1})$. If $f(s) = f^*(s)$, $\forall$ s $\in$ Pref($s_1...s_k s_{k+1}$), then:

a. There is no arc labelled ϕ_{k+1} emerging from q_{jk}^*.

b. $z_{ik} = z_{jk}^*$, $y_{ik} = y_{jk}^*$, $\mathcal{M}_{ik} \equiv \mathcal{M}_{jk}^*$.

Proof:

1. *a* - *d* follow by simultaneous induction on $r \in \{0, ..., k\}$. For $r = 0$, they are obvious. The induction step from r to r+1 is as follows.

Since *a* - *d* are true for r, the string $s_1...s_r$ will take $\mathcal{M}'$ and $\mathcal{M}^{*\prime}$ into the states (q_{ir}, p_{iro}) and (q_{jr}^*, p_{jro}^*) respectively. The corresponding memory states will be $(m_r, z_{ir}(m_r))$, $(m_r^*, z_{jr}(m_r^*))$, with $m_r = m_r^*$. Our strategy is to apply inputs to $\mathcal{M}'$ and $\mathcal{M}^{*\prime}$ in the above mentioned states and memory values.

Let $\sigma = head(s_{r+1})$. From the way in which s_{r+1} is chosen it follows that $\mathcal{M}'$ will perform a type A transition when it receives σ. Let $\gamma \in \Gamma_1$ be the output produced by $\mathcal{M}'$ when it receives σ. From the hypothesis, it follows that $\mathcal{M}^{*\prime}$ produces the same output γ when σ is applied. Now, we have two possible cases: σ will cause $\mathcal{M}^{*\prime}$ to perform a type A or a type B transition. If the transition was of type B, then we would have $\gamma \in \Gamma_2$. Since $\Gamma_1 \cap \Gamma_2 = \varnothing$, this is not possible. Therefore the transition is of type A. Since $\mathcal{R}$ is output-distinguishable, it follows that $\mathcal{M}_{ir} \equiv \mathcal{M}_{jr}^*$. Since $\mathcal{M}'$ is a simple refinement, we have $z_{ir} = z_{jr}^*$ and $y_{ir} = y_{jr}^*$.

We now apply s_{r+1} to $\mathcal{M}'$ in state (q_{ir}, p_{iro}) with memory value $(m_r, z_{ir}(m_r))$ and to $\mathcal{M}^{*\prime}$ in state (q_{jr}^*, p_{jro}) with memory value $(m_r^*, z_{jr}(m_r^*))$. Let g be the output sequence produced by $\mathcal{M}'$ and $\mathcal{M}^{*\prime}$ when they receive s_{r+1} (i.e. the fact that the two machines

produce the same output is guaranteed by the hypothesis). Then $g = g_1 \gamma_2$, with $g_1 \in \Gamma_1^*$, $|g_1| = |s_{r+1}| - 1$, and $\gamma_2 \in \Gamma_2$. Let us assume that there is no arc labelled ϕ_{r+1} emerging from q_{jr}^* in $\mathcal{M}^*$. Then, since $m_r = m_r^*$, $z_{ir} = z_{jr}^*$, $\mathcal{M}_{ir} \equiv \mathcal{M}_{jr}^*$, $y_{ir} = y_{jr}^*$, it follows that there is $\phi \in \Phi$, $\phi \neq \phi_{r+1}$, that produces the same output γ_2 as ϕ_{r+1} on the same input and memory value. This contradicts the output-distinguishability of the type Φ . Therefore, there is an arc labelled ϕ_{r+1} emerging from q_{jr}^*. Also $m_{r+1} = m_{r+1}^*$.

Therefore, we have proved the induction step for *a, b* and *d*. Clearly, *c* follows from these.

2. $z_{ik} = z_{jk}^*$, $y_{ik} = y_{jk}^*$, $\mathcal{M}_{ik} = \mathcal{M}_{jk}^*$ follow as above. Let us assume that there is an arc labelled ϕ_{k+1} emerging from q_{jk}^*. Then, using similar arguments as at 1, it follows that there exists an arc labelled ϕ_{k+1} emerging from q_{ik}, which contradicts our assumption. □

Before we proceed with the proof of our theorem we give the following definitions:

Definition 5.1.4.4.2.

A finite automata is called *accessible* if for all states q there exists a path that starts in the initial state of the automaton and ends in q.

Definition 5.1.4.4.3.

Let $\mathcal{A}$ and $\mathcal{A}^*$ two finite automata over alphabet Φ, Q and Q* their state sets and q_o and q_o^* their initial states respectively. Then g: $\mathcal{A} \rightarrow \mathcal{A}^*$ is called a *proper morphism* if g: $Q \rightarrow Q^*$ is a function that satisfy:

i) $g(q_o) = q_o^*$ and

ii) $q \xrightarrow{\phi} p$ is an arc in $\mathcal{A}$ iff $g(q) \xrightarrow{\phi} g(p)$ is an arc in $\mathcal{A}^*$, $\forall$ q, p $\in$ Q.

If g is bijective then g: $\mathcal{A} \rightarrow \mathcal{A}^*$ is called an *isomorphism*.

From the lemma above it follows that q_o and q_o^* are TZ-equivalent as states in $\mathcal{A}$ and $\mathcal{A}^*$ respectively (see Chow [1] or Ipate and Holcombe [6] for the definition of V-equivalent states, $V \subseteq \Phi^*$). From Chow [1], it follows that $\mathcal{A}$ and Min($\mathcal{A}^*$) are isomorphic, where $\mathcal{A}^*$ is the associated automaton of $\mathcal{M}^*$ and Min($\mathcal{A}^*$) is the minimised form of $\mathcal{A}^*$.

Without loss of generality we assume that $\mathcal{A}^*$ is accessible. Then

$$P = T(\Phi^k \cup \Phi^{k-1} \cup ... \cup \{\varepsilon\})$$

is a transition cover for $\mathcal{A}^*$. Indeed, since T is a transition cover for $\mathcal{A}$, there exists a state cover of $\mathcal{A}$, S, such that $T \supseteq S \cup S\Phi$. Then, since q_o and q_o^* are Φ^*-equivalent, it follows that at least card(Q) states of $\mathcal{A}^*$ will be accessed by some sequence in S. Since $\mathcal{A}^*$ is accessible and card(Q') $\leq$ card(Q) + k, it follows that

$$R = S(\Phi^k \cup \Phi^{k-1} \cup ... \cup \{\varepsilon\})$$

is a state cover of $\mathcal{A}^*$ (this can be proven easily using simple induction). Since $P \supseteq R \cup R\,\Phi$ it follows that P is a transition cover of $\mathcal{A}^*$.

Now, let g: $\mathcal{A}^* \rightarrow \mathcal{A}$ defined by $g(q_j^*) = q_i$ be such that q_i and q_j^* are Φ^*-equivalent. Since $\mathcal{A}$ is minimal, $\mathcal{A}^*$ is accessible and $\mathcal{A}$ and Min($\mathcal{A}^*$) are isomorphic it can be easily verified that g is well defined and it is a proper automata morphism. Then let $q_j^* \in Q^*$ and $q_i = g(q_j^*)$. Since $P = T(\Phi^k \cup \Phi^{k-1} \cup ... \cup \{\varepsilon\})$ is a transition cover for $\mathcal{A}^*$, there exist $\phi_1, ..., \phi_k, \phi_{k+1} \in \Phi$ such that $\phi_1...\phi_k, \phi_1...\phi_k\phi_{k+1} \in P$ and $\phi_1...\phi_k$ is the label of a path from q_o^* to q_j^*. Since $q_i = g(q_j^*)$, there also exists a path $\phi_1...\phi_k$ from q_o to q_i.

Since $f(s) = f^*(s)$, $\forall$ s $\in$ Pref(t(TZ)) it follows that

$$f(s) = f^*(s), \forall s \in Pref(\{\phi_1... \phi_{k+1}\}).$$

If $(z_i, y_i, \mathcal{M}_i) = Ref(q_i)$ and $(z_j^*\ y_j^*, \mathcal{M}_j^*) = Ref(q_j^*)$, using the above lemma, we have $z_i = z_j^*$, $y_i = y_j^*$, $\mathcal{M}_i \equiv \mathcal{M}_j^*$.

Therefore, there exists a proper morphism g: $\mathcal{A}^* \rightarrow \mathcal{A}$ with the following property:

$$\forall q_j^* \in Q^*, \text{ if } q_i = g(q_j^*) \text{ then } z_i = z_j^*, y_i = y_j^*, \mathcal{M}_i \equiv \mathcal{M}_j^*.$$

For $i \in \{0, ..., n\}$ and $j \in \{0, ..., n'\}$ let P_i and P_j^* be the state sets of $\mathcal{M}_i$ and $\mathcal{M}_j^*$, p_{if} and p_{jf}^* their final states and $\mathcal{A}_i$ and $\mathcal{A}_j^*$ their associated automata. Also, for $j \in \{0, ..., n'\}$ and $i \in \{0, ..., n\}$ such that $q_i = g(q_j^*)$, let h_j: $\mathcal{A}_j^* \rightarrow \mathcal{A}_i$ be the isomorphism between $\mathcal{A}_i$ and $\mathcal{A}_j^*$.

Then, the function

$$h: \bigcup_{j=0}^{n'} (\{q_j^*\} \times (P_j^* - p_{jf}^*)) \rightarrow \bigcup_{i=0}^{n} (\{q_i\} \times (P_i - p_{if}))$$

defined by

$$h(q_j^*, p_j^*) = (g(q_j^*), h_j(p_j^*))$$

is a proper morphism between the associated automata of $\mathcal{M}'^*$ and $\mathcal{M}'$ respectively. Hence, using a simple induction (see Ipate [5]) it follows that $\mathcal{M}'$ and $\mathcal{M}^{*\prime}$ compute the same function. □

5.2. The refinement testing method.

Our refinement testing method is based on the theorem above. It assumes that the following conditions are met:

1. The specification $\mathcal{M}'$ is a refinement of a deterministic stream X-machine $\mathcal{M}$ w.r.t. Ref, where Ref is a refinement function that takes values in the set $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}$;

2. The set of basic functions Φ of the control machine $\mathcal{M}$ is output-distinguishable;

3. The refinement set $\mathcal{R}$ is output-distinguishable;

4. The associated automaton $\mathcal{A}$ of $\mathcal{M}$ is minimal;

5. The refinement $\mathcal{M}'$ is simple and complete;

6. The output-alphabets of the control machine $\mathcal{M}$ and that of the basic modules $\mathcal{M}_i$ are disjoint (i.e. $\Gamma_1 \cap \Gamma_2 = \varnothing$)

7. The implementation can be modelled as a refinement of deterministic stream X-machine $\mathcal{M}^*$ w.r.t. Ref*, where Ref* takes values in the same refinement set $\mathcal{R}$, as Ref. Furthermore $\mathcal{M}$ and $\mathcal{M}^*$ have the same type Φ.

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

· t is a fundamental test function of the refinement $\mathcal{M}'$

· T is a transition cover of $\mathcal{A}$

· W is a characterisation set of $\mathcal{A}$

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

· k = card(Q*) - card(Q) is the difference between the (estimated) maximum number of states of $\mathcal{M}$* and the number of states of $\mathcal{M}$.

Obviously, the method relies on the system being specified as a refinement of a stream X-machine. Conditions 2 - 6 lie within the capability of the designer. The completeness of the refinement can be achieved by adding new inputs to the alphabet Σ and expanding $\mathcal{M}$ and $\mathcal{M}_i$ in a straightforward manner (see Ipate [5]). The output-distinguishability of $\mathcal{R}$ can be achieved by a simple augmentation of the output alphabet Γ_1. Similarly, Γ_1 and Γ_2 can be transformed into disjoint alphabets. The fact that we require that the refinement is simple is not a major problem. Indeed, if there exist i and j such that $\mathcal{M}_i \equiv \mathcal{M}_j$ and $z_i \neq z_j$ or $y_i \neq y_j$, then we transform $\mathcal{M}_i$ and $\mathcal{M}_j$ into output-distinguishable modules.

The 7'th condition is the most problematical. As we have discussed at the beginning of this section, it relies on the implementation of the system being constructed from the following components.

· the *correct* implementations of the basic functions Φ;

· the *correct* implementations $\mathcal{I}_i$ of (z_i, y_i, $\mathcal{M}_i$). The implementations of the modules $\mathcal{M}_i$ can be tested using the stream X-machine testing method.

If these conditions are met, the test set given by this method will test the integration of the above mentioned components. This consists of two things: testing that the control machine of the implementation $\mathcal{M}^*$ is correct and testing that each state in this machine is refined by the appropriate module.

Therefore, if the system is to be tested completely, a two phase approach is required.

· First, the ϕ's and the modules $\mathcal{M}_i$ are implemented and tested separately. Alternatively, their individual testing can be assumed to be done if either the implementations are very simple or they are objects that the designer is confident are essentially fault-free (i.e. objects from a library, etc.)

· The integration testing is carried out using the method presented above. Notice that the changes in the specifications of the refinement modules and processing functions required by the 'design for testing' conditions can be easily translated into changes in the corresponding implementations (see Ipate [5]). As we have seen earlier on, these changes will involve augmenting the processing functions and the refinement modules. Therefore, extra bits of code will be added to the existing implementations. These can be removed once the testing has been completed.

The maximum number of the test sequences required is similar to that for the stream X-machine testing method presented in Ipate and Holcombe [6]. The total length of the set is bigger since the fundamental test function of the refinement uses a sequence of characters to exercise each ϕ instead of single characters. However, we could require that

each such sequence of inputs has the length less then a certain number r (obviously, in this case the same condition will be imposed on the sequence of characters required by the completeness of the refinement condition (see definition 5.1.2.1)). In this case, the maximum number of the total length of the set given by our method is at most r times the upper bound of the total length of the test set given by the testing method presented in Ipate and Holcombe [6]. Also, similarly to Ipate and Holcombe, if the complexity of each ϕ of the control machine is at most C, it can be shown that the complexity of the algorithm that generates the test set will be proportional to

$$C \cdot card(\Sigma)^r \cdot card(\Phi)^{k+1} \cdot card(Q)^2 \cdot (2 \cdot card(Q^*) - card(Q))$$

If r is sufficiently small, it could be possible to generate the test set automatically.

6. Conclusions

The method of refining stream X-machine presented in this paper has been tried on several case studies (see Fairtlough et al. [2], Ipate [5], Laycock [7]) and appears to provide a simple way of developing stream X-machine specifications in an intuitive manner. A specification constructed using this type of refinement can be tested in two ways. The first is to construct the refined machine explicitly and use the stream X-machine testing method presented in Ipate and Holcombe [6]. This is feasible when the number of states of the refined machine is not too large. The alternative approach is to implement the refinement modules and the basic functions separately and to test the integrated system. This approach is suitable when the number of states of the refined

machine is quite large or when the basic refinement modules can be implemented quite easily (i.e. they are standard procedures or objects from a library or can be obtained from these very easily). A prerequisite of this approach is that the implementations of refinement modules and the basic functions of the control machines must be fault-free. Therefore, these have to be tested separately before the integration testing is carried out. The stream X-machine testing method can be used to test the refinement modules and also the processing functions, if these are expressed in terms of some (lower level) machines. Also, some 'design for test' conditions have to be met.

Also, the method requires careful testing management. Extra code has to be added to the existing implementation and removed after the testing has been completed, so it is essential that these changes are kept track of. However, this is not a major drawback of the method, since it is common practice to modify a program in order to test it. The trouble is that in most cases the fact that the program has passed the test does not enable us to say to much about its correctness. The advantage of our method is that it guarantees the correctness of the implementation of the whole system given correct implementation of its basic components. Very few existing functional testing methods attempt to deal with the integration issue. Many work on the assumption that if a system contains several components (modules, procedures, etc.) and each of these are correct, then somehow the whole system will also be correct. For example, let us see how a category-partition method can be used to test an implementation of a stream X-machine refinement. The method will require the following steps:

· Testing the basic functions of the control machine and the refinement modules.

· Testing the refinement modules.

· Testing the control machine.

However, there is no guarantee that, if all these are correct, the whole system will also be correct. Furthermore, the test set of the control machine will be a set of sequences of inputs for this machine (i.e. a set in I^*), without any indication being given of how these can be obtained from the real inputs (i.e. sequences from Σ^*) by the refinement modules. This is another important drawback of the category-partition method.

Recently, The X-machine model has been used for specification and testing in several case studies including safety-critical systems (see Holcombe et al. [4]) and the results are encouraging. However, if X-machine is to become a method that can be used widely in practice more ways of refining specifications and to generate their corresponding testing methods have to be introduced. Currently, there is work in progress which seeks to address these issues.

References.

[1] T. S. Chow, Testing software design modelled by finite state machines, *IEEE Transactions on Software Engineering*, **4**(3), 1978, pp. 178-187.

[2] M. Fairtlough, M. Holcombe, F. Ipate, C. Jordan, G. Laycock, Z. Duan, Using an X-machine to Model a Video Cassette Recorder, *Current issues in electronic modelling*, 3, 1995, pp. 141-161.

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

[4] M. Holcombe, F. Ipate, A. Grondoudis, Complete Functional Testing of Safety-Critical Systems, in *Proceedings of Second IFAC Workshop on Safety and Reliability in Emerging Control Technologies*, Daytona Beach, Florida, USA, 1-3 November 1995.

[5] F. Ipate, Theory of X-machines and Applications in Specification and Testing, PhD thesis, University of Sheffield, UK, 1995.

[6] F. Ipate. M. Holcombe. An integration testing method that is proved to find all faults, in *International Journal of Computer Mathematics*, 1997.

[7] G. T. Laycock, The theory and practice of specification based testing, Ph.D. thesis, University of Sheffield, UK, 1995.