Chapter 5.

Refinement of stream X-machines.

This chapter presents the concept of refinement as a way of developing stream X-machine specifications gradually. The concept will be illustrated with a case study (an X-machine specification of a word processor). A refinement testing method will also be given.

5.1. Refinement - definitions.

The concept of refinement we are presenting here provides a way of developing (generalised) stream X-machine specifications gradually. Simpler machines are used to construct a more complex specification. The way in which these 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. Let us explain now how our refinement works.

Let

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

be deterministic generalised stream X-modules (P_i is the state set, p_{io} is the initial state and all the 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 $p_{if} \in P_{io}$, $p_{io} \neq p_{if}$, 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.

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

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

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 5.1). 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 $g_1 \in \Gamma_1^*$ produced by $\mathcal{M}_i$. In this case we say that the whole system performs a *type A* transition (see figure 5.2).

ii) 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 $\mathcal{M}$ 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 $g_1\gamma_2$, where $g_1 \in \Gamma_1^*$ is the output produced by $\mathcal{M}_i$ when σ is received in p_i and m_i and $\gamma_2 \in \Gamma_2$ 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 5.2).

More formally, we have the following definition.

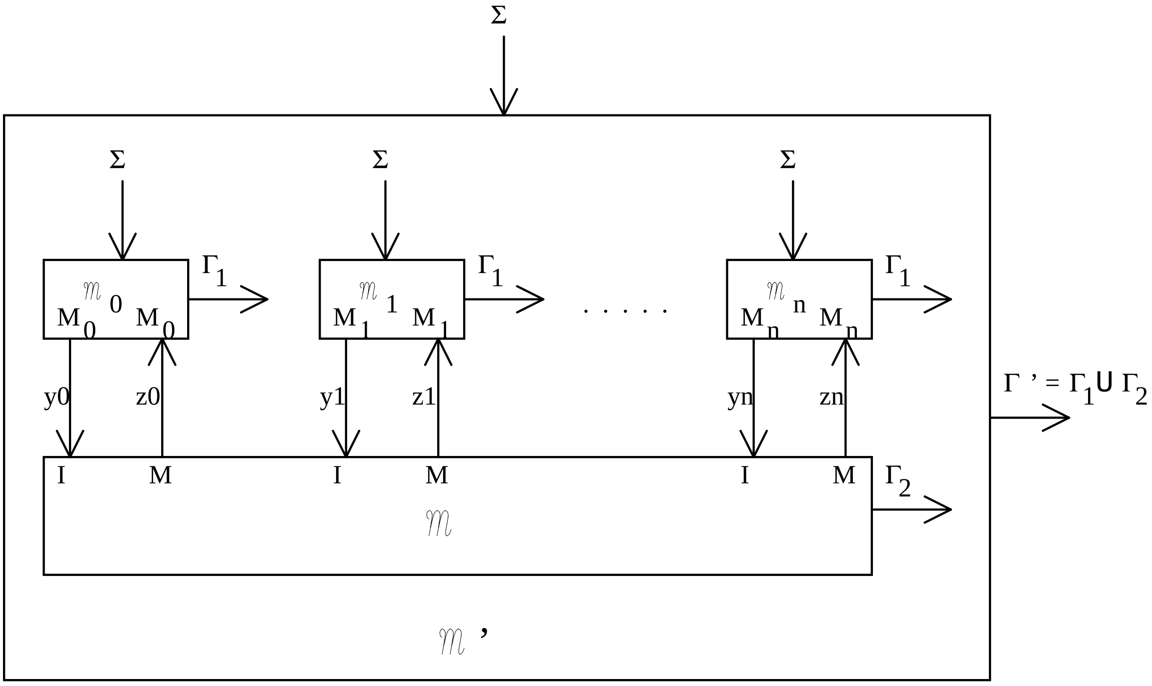


Figure 5.1.

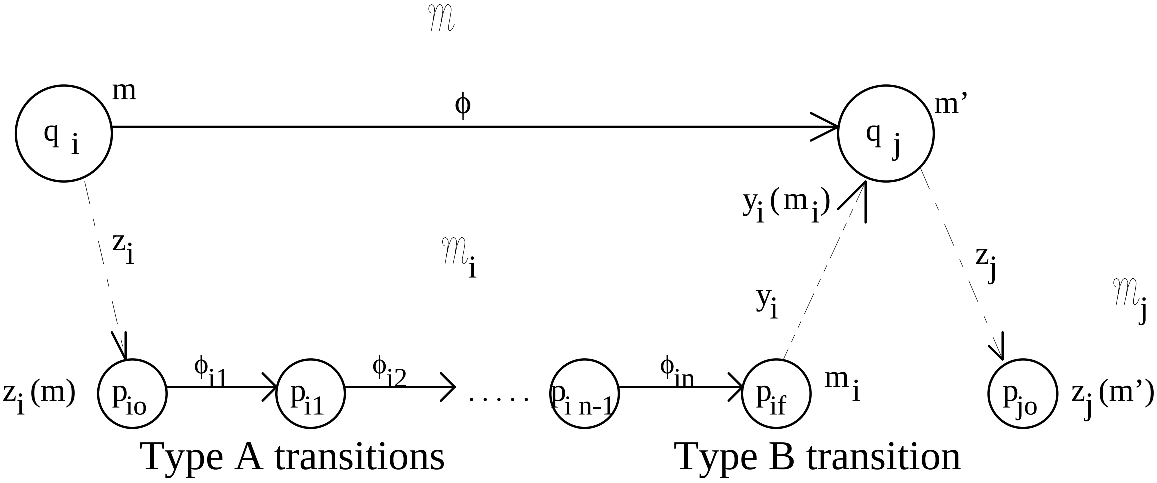


Figure 5.2.

Definition 5.1.1.

Let $\mathcal{M}_i = (\Sigma, \Gamma_1, P_i, M_i, \Phi_i, F_i, p_{io})$ be deterministic generalised stream X-modules as above and z_i: $M \rightarrow M_i$, y_i: $M_i \rightarrow I$ 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*: $Q \rightarrow \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a function defined by $Ref(q_i) = (z_i, y_i, \mathcal{M}_i)$. We define a generalised stream X-machine

$$\mathcal{M}' = (\Sigma, \Gamma', Q', M', q_0', m_o', \Phi', F')$$

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 $q_o' = (q_0, p_{oo})$.
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

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

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

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

$$c(\phi_i): M' \times \Sigma \rightarrow \Gamma'^* \times M'$$

is a (partial) function defined by:

$$c(\phi_i)((m, m_a), \sigma) = \begin{cases} (g_1, (m, m_a')), \text{ if } m_a \in M_i \text{ and } (m_a, \sigma) \in \text{dom } \phi_i \\ \emptyset, \text{ otherwise} \end{cases}$$

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

where

$$g_1 \in \Gamma_1^*, m_a' \in M_a \text{ satisfy } (g_1, m_a') = \phi_i(m_a, \sigma).$$

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). 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 $g_1 \in \Gamma_1{}^*$ and $m_i' \in M_i$ such that

$$(g_1, m_i') = \phi_i(m_a, \sigma).$$

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

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

Then

$$d(q_i, \phi, \phi_i)((m, m_a), \sigma) = \begin{cases} (g_1\gamma_2, (m', m_a')), & \text{if } m_a \in M_i,\ (m_a, \sigma) \in \text{dom } \phi_i \\ & \quad \text{and } (m, y_i(m_i')) \in \text{dom } \phi \\ \varnothing, & \text{otherwise} \end{cases}$$

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

where

$g_1 \in \Gamma_1{}^*$, $\gamma_2 \in \Gamma_2$, $m' \in M$, $m_i' \in M_i$ are defined as above and
$m_a' = z_j(m')$,
where j is chosen such that $q_j = F(q_i, \phi)$.

If $d(q_i, \phi, \phi_i)$ is the empty function, then $d(q_i, \phi, \phi_i)$ is not included in Φ_2'.

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:

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

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

Before we proceed further, we make the following remarks.

Observations.

1. Let $q_i \in Q$ and $p_i \in P_i - \{p_{if}\}$. Then it can be easily verified that:

i) $\forall\ \phi, \phi''\in \Phi, \phi_i, \phi_i'' \in \Phi_i$, if $d(q_i, \phi, \phi_i) = d(q_i, \phi'', \phi_i'') \in \Phi_2$’ and $F(p_i, \phi_i) \neq \emptyset$, $F(p_i, \phi_i'') \neq \emptyset$ then $\phi = \phi''$ and $\phi_i = \phi_i''$ (this is because $\mathcal{M}$ and $\mathcal{M}_i$ are deterministic).

ii) $\forall\ \phi_i, \phi_i'' \in \Phi_i$, if $c(\phi_i) = c(\phi_i'') \in \Phi_1$’, then$\phi_i = \phi_i''$.

iii) $\neg\exists\ \phi \in \Phi, \phi_i, \phi_i'' \in \Phi_i, \phi_i \neq \phi_i''$, that satisfy $d(q_i, \phi, \phi_i) = c(\phi_i'')$ and $F(p_i, \phi_i) \neq \emptyset$, $F(p_i, \phi_i'') \neq \emptyset$ (this is because $\mathcal{M}$ and $\mathcal{M}_i$ are deterministic).

Hence F'' is well defined.

2. $\mathcal{M}$' is deterministic (i.e. this follows from the fact that $\mathcal{M}$ and $\mathcal{M}_i$, i = 0, ..., n, are deterministic).

3. There may be elements of Φ' as defined above that do not actually appear in the state transition diagram of $\mathcal{M}$'. If we restrict Φ' only to those basic functions that are actually used by $\mathcal{M}$', we get

$$\Phi' = \Psi_1' \cup \Psi_2',$$

where

$$\Psi_1' = \{c(\phi_i) |\ \phi_i \in \Phi_i,\ i = 0, ..., n, \text{ such that } \exists\ p_i \in P_i \text{ with } F_i(p_i, \phi_i) \neq \emptyset \text{ and } F_i(p_i, \phi_i) \neq p_{if}\},$$

$$\Psi_2' = \{d(q_i, \phi, \phi_i) |\ i = 0,.., n,\ \phi \in \Phi,\ \phi_i \in \Phi_i \text{ such that } F(q_i, \phi) \neq \emptyset \text{ and } \exists\ p_i \in P_i \text{ with } F_i(p_i, \phi_i) = p_{if}\}.$$

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 memory is $M_i = \Sigma$.
3. The type is $\Phi_i = \{\phi_i\}$, with $\phi_i: \Sigma \times \Sigma \rightarrow \{1\} \times \Sigma$ a function defined by:

   $\phi_i(\sigma, \sigma') = (1, \sigma')\ \forall\ \sigma, \sigma' \in \Sigma$ (i.e. 1 is the empty sequence).
4. The state transition diagram determined by F_i consists of a single arc from p_{io} to p_{if} labelled ϕ_i (see figure 5.3).

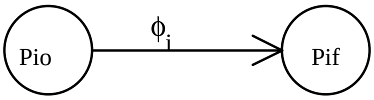


Figure 5.3.

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 → Σ 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: Σ → I will be an injective function that converts an input into the corresponding input *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.

If the refinement modules have the additional property that all the transitions that lead to a non-final state output a single character and that the remaining transitions produce no output at all, then the refined machine will be a stream X-machine rather then a generalised stream X-machine. This is formalised in what follows.

Definition 5.1.2.
Let $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a refinement set. Then $\mathcal{R}$ is called a *proper* refinement set if $\forall\ i \in I$, the type Φ_i of the module $\mathcal{M}_i$ can be written as

$$\Phi_i = \Phi_{i1} \cup \Phi_{i2}$$

such that:
i) $\forall\ \phi_i \in \Phi_{i1}$, Im $\phi_i \subseteq M_i \times \Sigma$;
ii) $\forall\ \phi_i \in \Phi_{i2}$, Im $\phi_i \subseteq M_i \times \{1\}$, where 1 is the empty string.
iii) $\forall\ \phi_i \in \Phi_i$ and $p_i \in P_i - \{p_{if}\}$ such that $F(p_i, \phi_i) \neq \emptyset$, $F(p_i, \phi_i) = p_{if}$ iff $\phi_i \in \Phi_{i2}$.

Lemma 5.1.3.
Let $\mathcal{M}$’ be a refinement of $\mathcal{M}$ w.r.t. Ref, where Ref: Q → $\mathcal{R}$ is the refinement function. If $\mathcal{R}$ is a proper refinement set, then $\mathcal{M}$’ is a stream X-machine.

Proof:
We saw that if we restrict the type of $\mathcal{M}$’ only to those basic functions that are used in the transition diagram of $\mathcal{M}$’, we get $\Phi' = \Psi_1' \cup \Psi_2'$ where Ψ_1' and Ψ_1' are defined above. It can be shown easily that any $\phi \in \Phi'$ outputs exactly one output symbol for each input symbol it receives. ©

In what follows we shall be referring mainly to the case in which the refinement set is proper (and therefore the resulting machine will be a stream X-machine).

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 example 5.1.6). 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.

Before we proceed with an example, we show that the construction made in definition 5.1.1 fits the informal description of refinement given at the beginning of the chapter.

Lemma 5.1.4.
Let $\mathcal{M}'$ be the refinement of $\mathcal{M}$ w.r.t. Ref as defined above. Let u, u', u_i, the transition functions and λ, λ', λ_i the output functions of $\mathcal{M}$, $\mathcal{M}'$ and $\mathcal{M}_i$ respectively. Then $\forall\, i \in \{0, ..., n\}, \sigma \in \Sigma, m \in M, m_a \in M_a, p_i \in P_i - \{p_{if}\}$ we have:

1. If $m_a \notin M_i$ or $u_i(p_i, m_a, \sigma) = \emptyset$, then
   $u'((q_i, p_i), (m, m_a), \sigma) = \emptyset$ and $\lambda'((q_i, p_i), (m, m_a), \sigma) = \emptyset$.

2. Otherwise let $u_i(p_i, m_i, \sigma) = (p_i', m_i')$ and $\lambda_i\,(p_i, m_i, \sigma) = g_1$, with $p_i' \in P_i$, $m_i' \in M_i$, $g_1 \in \Gamma_1{}^*$. Then:

a. If $p_i' \neq p_{if}$, then
$u'((q_i, p_i), (m, m_a), \sigma) = ((q_i, p_i'), (m, m_i'))$ and $\lambda'((q_i, p_i), (m, m_a), \sigma) = g_1$ (i.e. this is a type A transition)

b. If $p_i' = p_{if}$, then $u'((q_i, p_i), (m, m_a), \sigma) \neq \emptyset$ iff $u(q_i, m, y_i(m_i')) \neq \emptyset$. If this is the case let $u(q_i, m, y_i(m_i')) = (q_j, m')$ and $\lambda(q_i, m, y_i(m_i')) = \gamma_2$, with $m' \in M$,
$\gamma_2 \in \Gamma_2$. Then
$u'((q_i, p_i), (m, m_a), \sigma) = ((q_j, p_{jo}), (m', z_j(m')))$ and
$\lambda'((q_i, p_i), (m, m_a), \sigma) = g_1\gamma_2$
(i.e. this a type B transition).

Proof:
It follows from the construction of $\mathcal{M}'$ (the definitions ofΦ' and F').⑥

Notice that if the refinement set is proper then $g_1 \in \Gamma_1$ for type A transitions and $g_1 = 1$ for type B transitions.

Lemma 5.1.5.
Let $\mathcal{M}$' be the refinement of $\mathcal{M}$ w.r.t. Ref as defined above. Let u be the transition function and λ the output function of $\mathcal{M}$. Let also u_e', u_{ie} be the extended transition functions and λ_e', λ_{ie} the extended output functions of $\mathcal{M}$' and $\mathcal{M}_i$ respectively. Let $i \in \{0, ..., n\}$, $s \in \Sigma^*$, $m \in M$, $m_a \in M_i$, $p_i \in P_i - \{p_{if}\}$ such that $u_{ie}(p_i, m_a, s) \neq \varnothing$ and let $u_{ie}(p_i, m_a, s) = (p_i', m_i')$ and $\lambda_{ie}(p_i, m_a, s) = g_1$ with $p_i' \in P_i$, $m_i' \in M_i$, $g_1 \in \Gamma_1^*$. Then:

1. If $p_i' \neq p_{if}$, then

   $u_e'((q_i, p_i), (m, m_a), s) = ((q_i, p_i'), (m, m_i'))$ and

   $\lambda_e'((q_i, p_i), (m, m_a), s) = g_1$.

2. If $p_i' = p_{if}$, then $u_e'((q_i, p_i), (m, m_a), s) \neq \varnothing$ iff $u(q_i, m, y_i(m_i')) \neq \varnothing$. If this is the case, let $u(q_i, m, y_i(m_i')) = (q_j, m')$ and $\lambda(q_i, m, y_i(m_i')) = \gamma_2$, with $m' \in M$, $\gamma_2 \in \Gamma_2$. Then

   $u_e'((q_i, p_i), (m, m_a), s) = ((q_j, p_{jo}), (m', z_j(m')))$ and

   $\lambda_e'((q_i, p_i), (m, m_a), s) = g_1\gamma_2$.

Proof:
It follows by induction on s. ⑥

So, if s takes the module $\mathcal{M}_i$ into a state p_i' which is not the final state of the module, then the corresponding transitions caused by s in the machine $\mathcal{M}$' will all be of type A. If p_i' is the final state of the module, then $\mathcal{M}$' will perform |s|-1 type A transitions followed by a transition of type B. If the refinement set is proper, then $|g_1| = |s|$ if all the transitions performed are of type A and $|g_1| = |s|-1$ if the machine performs a sequence of type A transitions followed by a type B transition.

Example 5.1.6.

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. We require that each user can enter his/her password twice. Once the user has entered the system, the program only allows him/her to exit the system; any other command will be ignored.

The inputs used are: character keys, the Enter key and the Backspace key. Only usernames and passwords of at most n characters will be considered valid. Thus, if more than n characters have been entered, the rest will be ignored, unless one or more of the first n 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.
The number of characters in a command is unlimited.

5.1.6.1. 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).

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

1. The input set is I = STRINGS,

where

STRINGS = CHARACTERS* is the set of all sequences of characters.

2. The output is Γ_2 = MESSAGES is a set of messages or sequences of messages,

MESSAGES = {`msg1`, ..., `msg6`},

where:

`msg1` = ’insert your password:’,
`msg2` = ’login successful’,
`msg3` = ’wrong password□ insert your password:’,
`msg4` = ’login incorrect□ login:’,
`msg5` = ’exit system□ login:’,
`msg6` = ’unknown command’.

3. The set of states is:

Q = {Await_name, Await_psw1, Await_psw2, Await_command}.

The initial state is Await_name.

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

STRINGS$_n$ = $\sum_{k=o}^{n}$ CHARACTERS^k is the set of sequences of at most n characters.

We assume that the system keeps a data structure (i.e. *acc* ∈ ACCOUNT_INFO) for all the existing usernames and their associated passwords. As for the cash machine example, we do not make any assumptions about the way in which this is implemented. Instead, we assume that it can be manipulated via the following (partial) functions:

· `name_found`: ACCOUNT_INFO × STRINGS$_n$ → B (function), where B is the set of Booleans.

i.e. `name_found`(*acc, str*) is true if *str* is a valid username and false otherwise .

· `get_psw`: ACCOUNT_INFO × STRINGS$_n$ → STRINGS$_n$ (partial function)

i.e. this finds the corresponding password of a username if the username is valid.

The (partial) functions above satisfy:

dom `get_psw` =
$\{(acc, str) \in$ ACCOUNT_INFO × $\text{STRINGS}_n \mid$ `name_found`$(acc, str)\}$

The system memory will be a tuple (*acc*, *mem_str*), where *acc* ∈ ACCOUNT_INFO and *mem_str* ∈ STRINGS_n stores the last username that has been entered.

5. The initial memory value is

m_o = (*`in_acc`*, empty_seq),

where *in_acc* is the initial value of ACCOUNT_INFO and empty_seq denotes the empty string (i.e. for the sake of clarity the empty string will be denoted by empty_seq instead of 1).

6. The type is:

Φ = {`enter_name`, `good_psw`, `wrong_psw1`, `wrong_psw2`, `ignore_command`, `exit_system`}.

7. The state transition diagram is represented in figure 5.4.

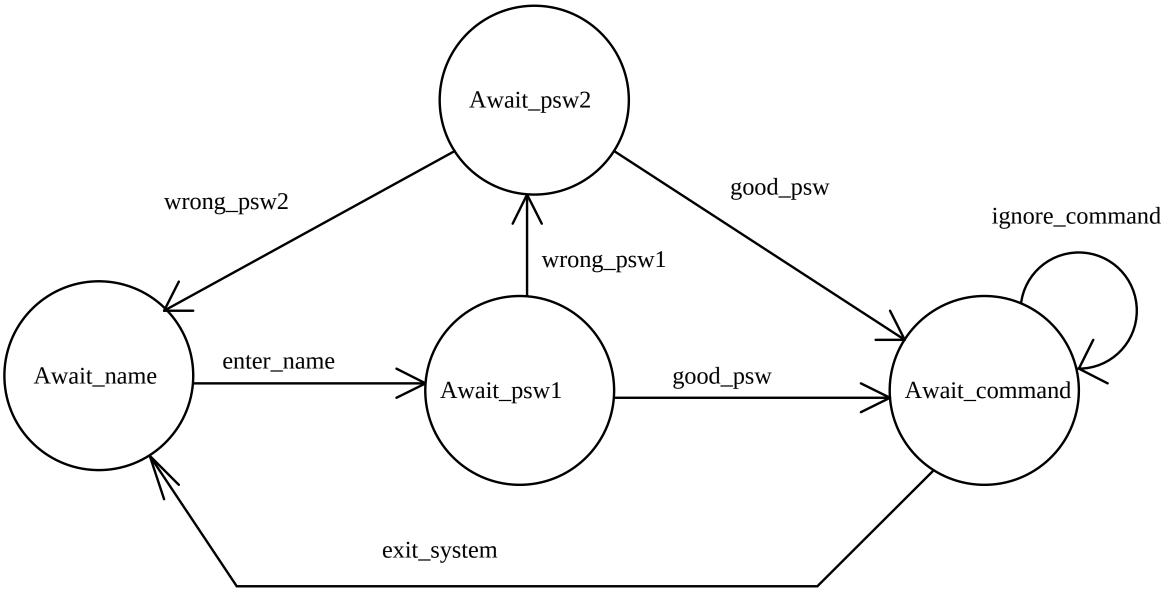


Figure 5.4.

8. The basic processing functions are defined as follows:

· dom `enter_name` = M × STRINGS_n

`enter_name`((*acc*, *mem_str*), *str*)) = (*`msg1`*, (*acc*, *str*))

i.e. the name entered (i.e. *str* ∈ STRINGS_n) is copied into the appropriate memory register.

· dom `good_psw` = {(*acc*, *mem_str*), *str*) ∈ M × STRINGS_n| `name_found`(*acc*, *mem_str*) and `get_psw`(*acc*, *mem_str*) = *str*}

`good_psw`((*acc*, *mem_str*), *str*) = (*msg2*, (*acc*, *mem_str*))

i.e. this function is applied if the password matches the correct one.

· dom `wrong_psw1` = {(*acc*, *mem_str*), *str*) ∈ M × STRINGS_n|
¬(`name_found`(*acc*, *mem_str*) and `get_psw`(*acc*, *mem_str*) = *str*)}

`wrong_psw1`((*acc*, *mem_str*), *str*) = (*msg3*, (*acc*, *mem_str*))

· dom `wrong_psw2` = {(*acc*, *mem_str*), *str*) ∈ M × STRINGS_n|
¬(`name_found`(*acc*, *mem_str*) and `get_psw`(*acc*, *mem_str*) = *str*)}

`wrong_psw2`((*acc*, *mem_str*), *str*) = (*msg4*, (*acc*, *mem_str*))

i.e. these functions are applied when the password does not match the correct one.

· dom `ignore_command` = M × (STRINGS - {exit})
`ignore_command`((*acc*, *mem_str*), *str*) = (*msg5*, (*acc*, *mem_str*))

i.e. if *str* ≠ ’exit’, the command is ignored.

· dom `exit_system` = M × {’exit’}
`exit_system`((*acc*, *mem_str*), ’exit’) = (*msg6*, (*acc*, empty_seq))

i.e. otherwise, the system returns to its initial state; the memory register that holds the last username entered becomes empty.

5.1.6.2. The refinement function.

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

Ref: {Await_name, Await_psw1, Await_psw2, Await_command} → $\{(z_i, y_i, \mathcal{M}_i)\}_{i=0,...3}$

defined by:

Ref(Await_name) = $(z_0, y_0, \mathcal{M}_0)$,
Ref(Await_psw1) = $(z_1, y_1, \mathcal{M}_1)$,
Ref(Await_psw2) = $(z_2, y_2, \mathcal{M}_2)$,
Ref(Await_command) = $(z_3, y_3, \mathcal{M}_3)$.

5.1.6.2.1. The refinement modules.

$\mathcal{M}_0$, $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$ are generalised 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`},

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

Then, the four 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$_n$.

3. The type is:

$\Phi_0 = \Phi_{01} \cup \Phi_{02}$,

where

Φ_{01} = {`type_ch1`, `type_ch2`, `press_bs1`, `press_bs2`}

Φ_{02} = {`press_enter1`}

4. The transition diagram is shown in figure 5.5.

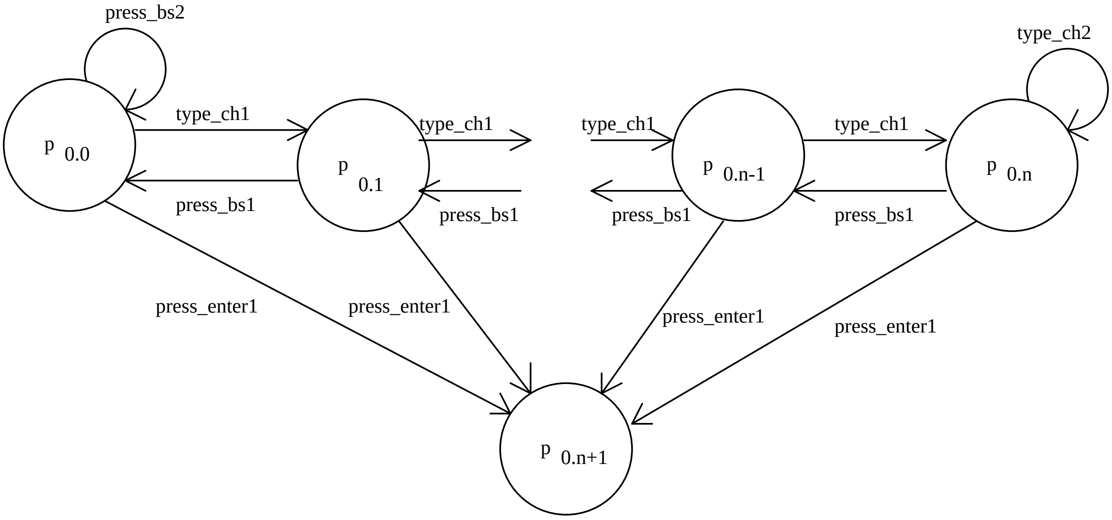


Figure 5.5.

5. The basic functions are defined by:

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

`type_ch1`(*str*, *ch*) = (`display`(*ch*), *str ch*)

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

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

`type_ch2`(*str*, *ch*) = (***`empty_display`***, *str*)

i.e. if the machine is in the state $p_{0.n}$ (therefore $|str| = n$) then nothing is displayed and the memory value (i.e. *str*) remains unchanged.

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

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

i.e. the last character is removed from the screen and the sequence *str*.

· dom `press_bs2` = $\text{STRINGS}_n \times$ {*`back_space`*}

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

i.e. if the machine is in the state $p_{0.0}$ (therefore *str* = empty_seq), then nothing is displayed and the current memory value remains unchanged.

· dom `press_enter1` = $\text{STRINGS}_n \times$ {*`enter`*}

`press_enter1`(*str*, *`enter`*) = (empty_seq, *str*)

i.e. when *`enter`* is pressed, the machine goes to the final state $p_{0.n+1}$.

B. $\mathcal{M}_1$.

1. The state set is $P_1 = \{p_{1.0}, ..., p_{1.n}, p_{1.n+1}\}$;
$p_{1.0}$ is the initial state; $p_{1.n+1}$ is the final state.
2. $M_1 = \text{STRINGS}_n$.
3. The type is:

$$\Phi_1 = \Phi_{11} \cup \Phi_{12},$$

where

Φ_{11} = {`type_ch2`, `type_ch3`, `press_bs2`, `press_bs3`},

Φ_{12} = {`press_enter1`}

4. The transition diagram is shown in figure 5.6.

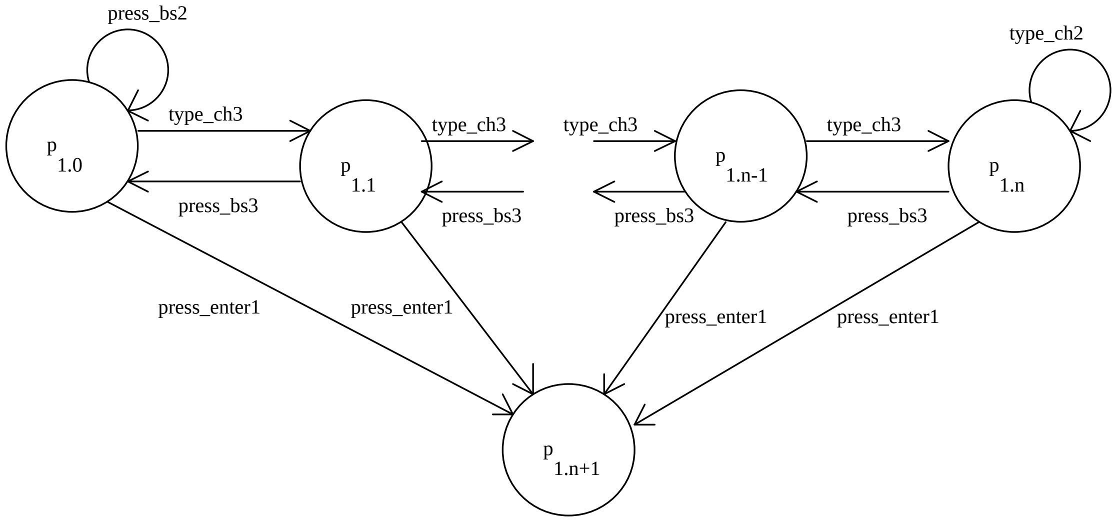


Figure 5.6.

5. The transition functions `type_ch3` and `press_bs3` are defined as follows:

· dom `type_ch3` = $\text{STRINGS}_n \times$ CHARACTERS

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

· dom `press_bs3` = $M_1 \times$ {*`back_space`*}

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

The definitions above are similar to those of `type_ch1` and `press_bs1` apart from the fact that nothing is displayed.

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

D. $\mathcal{M}_3$.

1. The set of states is $P_3 = \{p_{3.0}, p_{3.1}\}$;

$p_{3.0}$ is the initial state; $p_{3.1}$ is the final state;

2. M_3 = STRINGS.

3. The type is:

$\Phi_3 = \Phi_{31} \cup \Phi_{32}$,

where

Φ_{31} = {`type_ch4`, `press_bs4`},

Φ_{32} = {`press_enter2`}

4. The transition diagram is shown in figure 5.7.

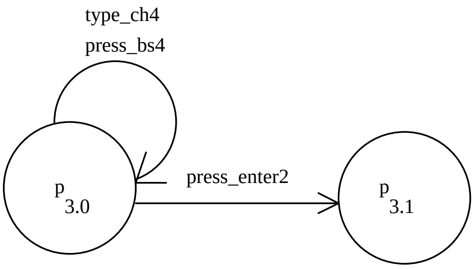


Figure 5.7.

5. The basic functions are defined by:

· dom `type_ch4` = STRINGS × CHARACTERS

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

i.e the character *ch* is displayed and added at the end of *str* ∈ STRINGS.

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

`press_bs4`(*str*, *`back_space`*) = (*displ*, tail(*str*))

where:

$$displ = \begin{cases} \mathit{delete\_char}, \text{ if } str \neq \text{empty\_seq} \\ \mathit{empty\_display}, \text{ if } str = \text{empty\_seq} \end{cases}$$

i.e. if, *str* ≠ empty_seq, then the last character is removed from the screen and the sequence *str*. Otherwise, nothing is displayed and the current memory value remains unchanged.

· dom `press_enter2` = STRINGS × {*`enter`*}

`press_bs4`(*str*, *`enter`*) = (empty_seq, *str*)

i.e. when *`enter`* is pressed, the machine goes to the final state $p_{3.n+1}$.

5.1.6.2.2. The transfer functions.

For i = 0, ..., 2, z_i: M → STRINGS_n and y_i: STRINGS_n → STRINGS are defined by:

$z_i(m)$ = empty_seq, ∀ m ∈ M,
$y_i(\mathit{str}) = \mathit{str}$, ∀ *str* ∈ STRINGS_n.

z_3: M → STRINGS and y_3:STRINGS → STRINGS are defined by:

$z_i(m)$ = empty_seq, ∀ m ∈ M,
$y_i(\mathit{str}) = \mathit{str}$, ∀ *str* ∈ STRINGS.

5.1.6.3. 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 Γ’ =$\Gamma_1 \cup \Gamma_2$. Therefore
   Γ’ = DISPLAYS∪ {*delete_char*, *empty_display*} ∪ MESSAGES.

3. The state set is
   Q’ = {$q_{0.0}$, ..., $q_{0.n}$, $q_{1.0}$, ..., $q_{1.n}$, $q_{2.0}$, ..., $q_{2.n}$, $q_{3.0}$}

(i.e. $q_{0.0}$ = (Await_name, $p_{0.0}$) ..., $q_{0.n}$ = (Await_name, $p_{0.n}$),
$q_{1.0}$ = (Await_psw1, $p_{1.0}$), ..., $q_{1.n}$ = (Await_psw1, $p_{1.n}$),
$q_{2.0}$ = (Await_psw2, $p_{1.0}$), ..., $q_{2.n}$ = (Await_psw2, $p_{1.n}$),
$q_{3.0}$ = (Await_command, $p_{3.0}$)).

The initial state is $q_{0.0}$.

4. The memory is M’ = ACCOUNT_INFO× STRINGS_n × STRINGS

(i.e. M’ = M × M_a, where M = ACCOUNT_INFO × STRINGS_n and M_a = STRINGS).

The initial memory value is
m_o’ = (*in_acc*, empty_seq, empty_seq).

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_enter1.3′, press_enter1.4′, press_enter2.1′, press_enter2.2′},

where

`type_ch1′` = *c*(`type_ch1′`1), ..., `type_ch4′` = *c*(`type_ch4`), ...

`press_bs1′` = *c*(`press_bs1`), ..., `press_bs4′` = *c*(`press_bs4`)

`press_enter1.1′` = *d*(Await_name, `enter_name`, `press_enter1`),

`press_enter1.2′` = *d*(Await_psw1, `good_psw`, `press_enter1`)
= *d*(Await_psw2, `good_psw`, `press_enter1`),

`press_enter1.3′` = *d*(Await_psw1, `wrong_psw1`, `press_enter1`)

`press_enter1.4′` = *d*(Await_psw2, `wrong_psw2`, `press_enter1`),

`press_enter2.1′` = *d*(Await_command, `exit_system`, `press_enter2`)

`press_enter2.2′` = *d*(Await_command, `ignore_command`,
`press_enter2`)

(i.e. the other functions obtained by applying *c* and *d* will not appear in the transition diagram).

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*, *mem_str1*, *mem_str2*), *ch*) =
(`display`(*ch*), (*acc*, *mem_str1*, *mem_str2 ch*))

· dom `press_enter1.2′` = {((*acc*, *mem_str1*, *mem_str2*), *`enter`*))| *acc*∈ ACCOUNT_INFO, *mem_str1*, *mem_str2* ∈ STRINGS_n such that (`name_found`(*acc*, *mem_str1*) and `get_psw`(*acc*, *mem_str1*) = *mem_str*2)};

`press_enter1.2′`((*acc*, *mem_str1*, *mem_str1*), *`enter`*) =
(*`msg2`*, (*acc*, *mem_str1*, empty_seq)) .

· dom `press_enter2.1′` = (ACCOUNT_INFO × STRINGS_n × {’exit’}) × {*`enter`*};

`press_enter2.1′`((*acc*, *mem_str1*, ’exit’),*`enter`*) =
(*`msg6`*, (*acc*, empty_seq, empty_seq)).

6. The state transition diagram is represented in figure 5.8.

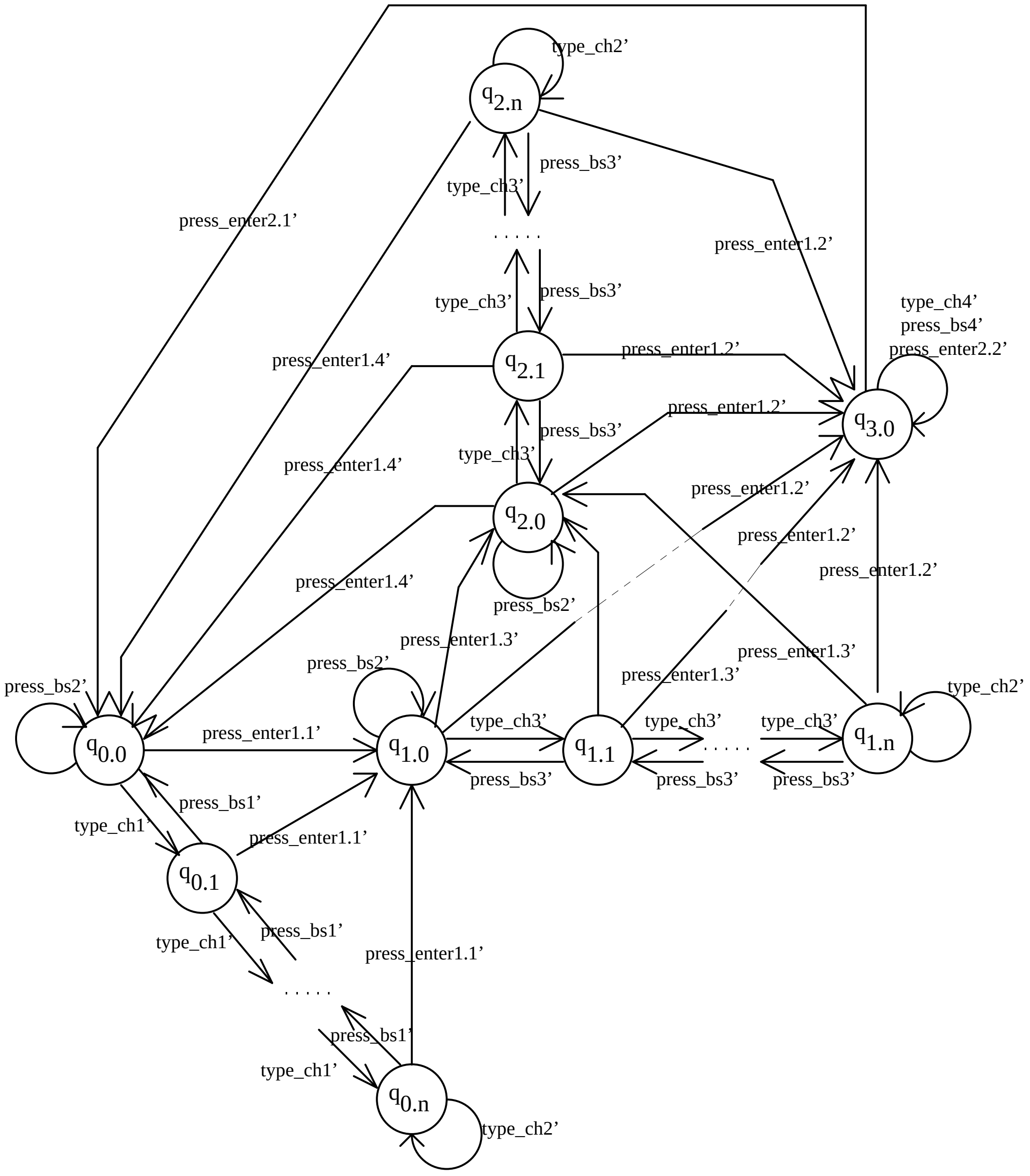

type_ch2'
$q_{2.n}$
press_bs3'
type_ch3'
press_enter2.1'
. . . . .
press_enter1.2'
press_bs3'
type_ch3'
type_ch4'
press_bs4'
press_enter2.2'
press_enter1.4'
$q_{2.1}$
press_enter1.2'
press_enter1.2'
press_bs3'
$q_{3.0}$
type_ch3'
press_enter1.4'
$q_{2.0}$
press_enter1.2'
press_enter1.2'
press_enter1.2'
press_enter1.4'
press_bs2'
press_enter1.3'
press_enter1.3'
press_enter1.3'
press_bs2'
press_bs2'
type_ch2'
type_ch3'
type_ch3'
type_ch3'
press_enter1.1'
$q_{0.0}$
$q_{1.0}$
$q_{1.1}$
. . . . .
$q_{1.n}$
press_bs3'
press_bs3'
press_bs3'
press_bs1'
type_ch1'
press_enter1.1'
$q_{0.1}$
press_bs1'
type_ch1'
press_enter1.1'
. . . . .
press_bs1'
type_ch1'
$q_{0.n}$
type_ch2'

Figure 5.8.

5.2. 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 $\mathcal{M}$.

5.3. 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 the previous chapter. 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 the previous chapter, we shall discuss now how the system integration can be tested.

As in previous chapter, 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.

As for the stream X-machine testing, the consecutive execution of two ϕ's or two $\mathcal{I}_i$'s has to be prevented (this can be done using a flag variable, that indicates whether the last piece of code executed by the program represented a ϕ or an $\mathcal{I}_i$).

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).

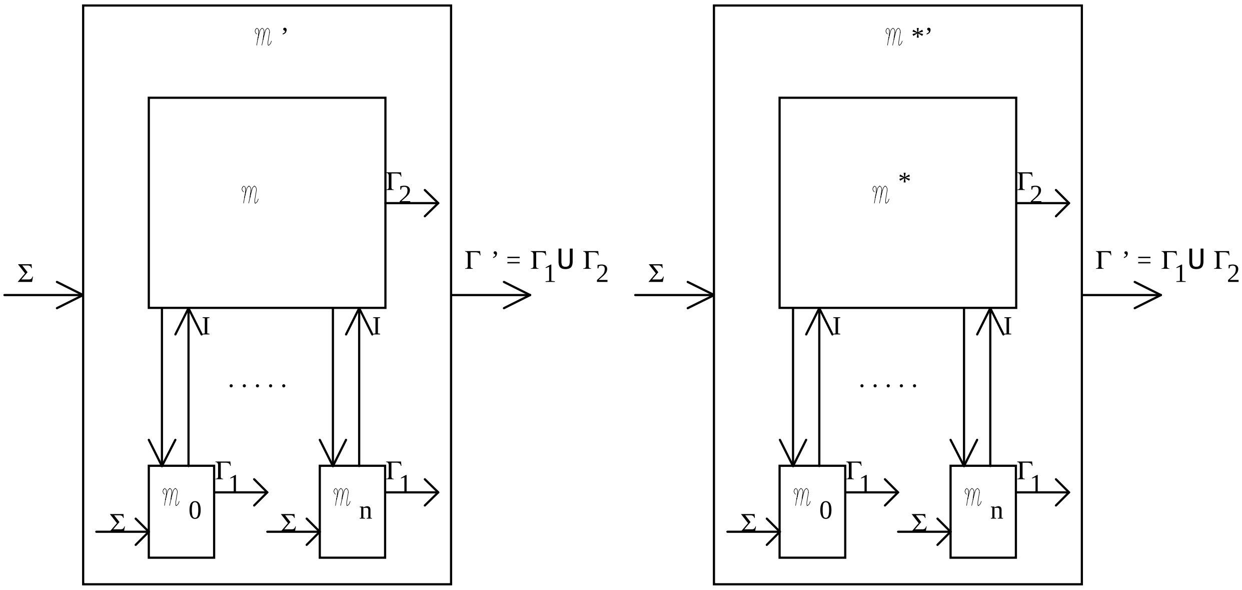
Therefore, the implementation of the whole system (let us call this $\mathcal{M}'$*) can be modelled as the refinement of $\mathcal{M}$* w.r.t. Ref*, where Ref* is a refinement function Ref*: Q* → $\{(z_i, y_i, \mathcal{M}_i)\}_{i=0,...,n}$, where Q* is the state set of $\mathcal{M}$* and $\{(z_i, y_i, \mathcal{M}_i)\}_{i=0,...,n}$ is the refinement set of the specification (see figure 5.9).

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. We shall assume that the refinement set is proper (and hence the resulting machine is a stream X-machine).



Specification Implementation

Figure 5.9.

5.3.1. ’Design for testing’ conditions.

Similar to the previous chapter, our specification has to satisfy some ’design for testing’ conditions.

5.3.1.1. Simple refinement.

Notation 5.3.1.1.1.
Let $\mathcal{M}_i$ and $\mathcal{M}_j$ be two (generalised) 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 5.3.1.1.2.
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 from example 5.1.6. is simple.

5.3.1.2. Complete refinement.

For similar reasons to those discussed in the previous chapter, our testing method requires some completeness and output-distinguishability conditions.

Definition 5.3.1.2.1.

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

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

$u_{ie}(p_{i0}, z_i(m), s) = (p_{if}, m_i)$

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_{i0} and memory value $z_i(m)$ to the final state p_{if} and m_i such that $(m, y_i(m_i)) \in \text{dom } \phi$.

Definition 5.3.1.2.2.

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

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

5.3.1.2.1. Enforcing the completeness condition.

This completeness condition can be enforced on a machine refinement by augmenting both Σ and I. A possible method is presented in what follows.

For i ∈ {1, ..., n} let

$\Xi_i = \{\phi \in \Phi | \ \mathcal{M}_i$ is not complete w.r.t. $\phi\}$

and $k_i = \text{card}(\Xi_i)$. Let also $\Xi = \bigcup_{i=0}^{n} \Xi_i$ and $k = \text{card}(\Xi)$.

Let in_1', ... in_k' and σ_1', σ_2' be new inputs such that $\{in_1', ...\ in_k'\} \cap I = \varnothing$ and $\{\sigma_1', \sigma_2'\} \cap \Sigma = \varnothing$. Let also

$a: \Xi \rightarrow \{in_1', ...\ in_k'\}$

and

$b_i: \Xi_i \rightarrow \{1, ..., k_i\}$, i = 0, ..., n,

be bijective functions. Then $\mathcal{M}$ and $\mathcal{M}_i$ are augmented as follows.

i) $\mathcal{M} = (I, \Gamma_2, Q, M, \Phi, F, q_0, m_0)$ is transformed into

$\mathcal{M}_A = (I_A, \Gamma_2, Q, M, \Phi_A, F_A, q_0, m_0)$

defined by:

1. The input set is $I_A = I \cup \{in_1', ...\ in_k'\}$.

2. The type Φ_A is defined by $\Phi_A = \{\phi_A | \ \phi \in \Phi\}$, where $\forall \phi \in \Phi$, $\phi_A: M \times I_A \rightarrow \Gamma_1 \times M$ is a (partial) function defined as follows.

· if $\phi \in \Phi - \Xi$, then

$\quad$ dom ϕ_A = dom ϕ and

$\quad$ $\phi_A(m, in) = \phi(m, in)$, $\forall$ $m \in M$, $in \in I$;

· if $\phi \in \Xi$, then

$\quad$ dom ϕ_A = dom $\phi \cup (M \times \{a(\phi)\})$ and

$\quad$ $\phi_A(m, in) = \phi(m, in)$, $\forall$ $m \in M$, $in \in I$,

$\quad$ $\phi_A(m, a(\phi)) = (\gamma_2, m)$, $\forall$ $m \in M$, where $\gamma_2 \in \Gamma_2$ is arbitrarily chosen.

3. The next state function will be defined by:

$\quad$ $F_A(q, \phi_A) = F(q, \phi)$, $\forall$ $\phi \in \Phi$.

Basically, the functions in Ξ are augmented by using an extra input for each of them. Everything else remains unchanged.

ii) $\mathcal{M}_i = (\Sigma, \Gamma_1, P_i, M_i, \Phi_i, F_i, p_{io})$ is transformed into

$\quad$ $\mathcal{M}_{iA} = (\Sigma_A, \Gamma_1, P_i, M_{iA}, \Phi_{iA}, F_{iA}, p_{io})$,

where:

1. The input set is $\Sigma_A = \Sigma \cup \{\sigma_1', \sigma_2'\}$.

2. The memory set is $M_{iA} = M_i \times \{0, ..., k_i\}$.

3. The type Φ_{iA} is defined by

$\quad$ $\Phi_{iA} = \{\phi_{iA} | \phi_i \in \Phi_i\} \cup \{\zeta_i, \xi_i\}$,

where:

· $\forall$ $\phi_i \in \Phi_i$, $\phi_{iA}: (M_i \times \{0, ..., k_i\}) \times \Sigma_A \rightarrow \Gamma_1{}^* \times (M_i \times \{0, ..., k_i\})$ is defined by:

$\quad$ dom $\phi_{iA} = \{((m_i, j), \sigma) | m_i \in M_i, \sigma \in \Sigma, j \in \{0, ..., k_i\}$ such that

$\quad\quad\quad$ $(m_i, \sigma) \in$ dom $\phi_i\}$;

$\quad$ $\phi_{iA}((m_i, j), \sigma) = ((m_i', j), g)$,

where $m_i' \in M_i$, $g \in \Gamma_1{}^*$ such that $(m_i', g) = \phi_i(m_i, \sigma)$.

· $\xi_i: (M_i \times \{0, ..., k_i\}) \times \Sigma_A \rightarrow \Gamma_1 \times (M_i \times \{0, ..., k_i\})$ is defined by:

$\quad$ dom $\xi_i = (M_i \times \{0, ..., k_i\}) \times \{\sigma_1'\}$;

$\quad$ $\xi_i((m_i, j), \sigma_1') = (\gamma_1, (m_i, j'))$, $\forall$ $m_i \in M_i$, $j \in \{0, ..., k_i\}$,

where $j' = (j + 1) \bmod k_i$ and $\gamma_1 \in \Gamma_1$ is arbitrarily chosen.

· $\zeta_i: (M_i \times \{0, ..., k_i\}) \times \Sigma_A \rightarrow \Gamma_1 \times (M_i \times \{0, ..., k_i\})$ is defined by:

$\quad$ dom $\zeta_i = (M_i \times \{0, ..., k_i\}) \times \{\sigma_2'\}$;

$\quad$ $\zeta_i((m_i, j), \sigma_2) = (1, (m_i, j))$, $\forall$ $m_i \in M_i$, $j \in \{0, ..., k_i\}$ (i.e. 1 is the empty string)

Basically, we augment the machine memory by adding an extra register which takes values in $\{0, ..., k_i\}$; the functions in Φ_i are essentially unaffected by this memory augmentation. The ξ_i changes only the value of this extra register, whereas the ζ_i leaves the entire memory unchanged.

3. The next state function F_{iA} is defined by

$$\mathrm{dom}\ F_{iA} = \{(p_i, \phi_{iA}) \in P_i \times \Phi_{iA} | (p_i, \phi_i) \in \mathrm{dom}\ F_i\} \cup$$
$$\{(p_i, \xi_i) | p_i \in P_i - \{p_i\}\} \cup \{(p_i, \zeta_i) | p_i \in P_i - \{p_i\}\};$$
$$F_{iA}(p_i, \phi_{iA}) = F_i(p_i, \phi_i),\ \forall\ p_i \in P_i,\ \phi_i \in \Phi_i,$$
$$F_{iA}(p_i, \xi_i) = p_i\ \forall\ p_i \in P_i,$$
$$F_{iA}(p_i, \zeta_i) = p_{if}\ \forall\ p_i \in P_i.$$

For each state $p_i \in P_i - \{p_{if}\}$, we add two extra arcs: one from p_i to itself (this is labelled ξ_i), the other from p_i to the final state of $\mathcal{M}_i$, p_{if} (this is labelled ζ_i).

At the first sight, the process of augmentation of the refinement modules appears to be quite complicated. Indeed, we could think of many other simpler methods in terms of extra memory and extra processing functions needed. However, the advantage of this method is that the augmentation of the implementation can be translated in a straightforward manner into a change in the implementation. Indeed, let us assume that we have a program that implements $\mathcal{M}_i$. Then, what we really have to do in order to simulate the above augmentation is to define a new memory register with values in $\{0, ..., k_i\}$ and to add two new pieces of code after each ’read’ instruction in the program. The first implements ξ_i. This will be selected when the input read is σ_1’ and will only update the extra memory register. Everything else (including state variables) remains unchanged. The second implements ζ_i. This will be selected when the input read is σ_2’ and will cause the program to exit.

The transition functions z_i and y_i will be transformed into

$$z_{iA}: M \rightarrow M_{iA},\ y_{iA}: M_{iA} \rightarrow I_A$$

defined by:

$$z_{iA}(m) = (z_i(m), 0)),\ \forall\ m \in M;$$

$$y_{iA}(m_i, 0) = y_i(m),\ \forall\ m_i \in M_i;$$
$$y_{iA}(m_i, j) = a(b_i^{-1}(j)),\ \forall\ m_i \in M_i,\ j \in \{1, ..., k\}.$$

We can now show that $\forall\ \phi \in \Xi_i$, $\mathcal{M}_{iA}$ is complete w.r.t. ϕ. Indeed, let $\phi \in \Xi_i$ Then $\exists\ j \in \{1, ..., k\}$ and $l \in \{1, ..., k_i\}$ such that in_j’$= a(\phi)$ and $l = b_i(\phi)$. Then $\forall$ $m \in M$, $s = \sigma_1{}^l\ \sigma_2$ takes $\mathcal{M}_i$ from the initial state p_{io} and memory value $(z_i(m), 0)$ to p_{if} and $(z_i(m), l)$. Since $y_{iA}(z_i(m), l) = in_j$’ and $(m, in_j$’$) \in \mathrm{dom}\ \phi$, it follows that $\mathcal{M}_{iA}$ is complete w.r.t. ϕ.

Clearly $\mathcal{M}_{iA}$ is complete w.r.t. ϕ, $\forall\ \phi \in \Phi - \Xi_i$. Hence, the augmented refinement is complete.

Example 5.3.1.2.1.1.

Let $\mathcal{M}$' be the refinement from example 5.1.6. If we assume that $n \geq 4$ (i.e. the maximum allowed length of the usernames and passwords is no less then 4), then we have:

$\Xi_0 = \Xi_1 = \Xi_2 = \Xi_3 = \{\texttt{good\_psw}\}$.

Then

$\Xi = \{\texttt{good\_psw}\}$.

Observations: We assume that not all the user names that can be entered are valid, hence $\texttt{good\_psw} \in \Xi_i$, i = 0, ..., 3. Also, since n > 4, $\texttt{exit\_system} \notin \Xi_i$, i = 0, .., 2 (i.e. the length of the string 'exit' is 4).

Hence $k_0 = k_1 = k_2 = k_3 = 1$ and k = 1.

Then let in_1' and σ_1', σ_2' be the extra inputs required. The augmented machine $\mathcal{M}_A$ will have the input set $I_A = I \cup \{in_1'\}$; $\texttt{good\_psw}$ will be augmented to $\texttt{good\_psw}_A$, where:

dom $\texttt{good\_psw}_A = \texttt{good\_psw} \cup (M \times \{in_1'\})$.

The augmented modules $\mathcal{M}_{iA}$ will have the input set $\Sigma_A = \Sigma \cup \{\sigma_1', \sigma_2'\}$. The augmented memory sets will be $M_{0A} = M_{1A} = M_{2A} = STRINGS_n \times \{0, 1\}$ and $M_{3A} = STRINGS \times \{0, 1\}$.

The main difficulty in this method is determining the sets Ξ_i. In the worst case we can take $\Xi_i = \Phi$, i = 0, ..., n. In this case, the resulting augmented refinement will be complete irrespective of whether the $\mathcal{M}_i$'s are complete or not with any$\phi \in \Phi$.

5.3.1.3. Output-distinguishable refinement set.

The second condition required by our refinement testing method will be the output-distinguishability of the modules $\mathcal{M}_i$. In what follows we shall assume that the refinement set is proper. Recall that if the refinement set $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ is proper the the type of the module $\mathcal{M}_i$ can be written as $\Phi_i = \Phi_{i1} \cup \Phi_{i2}$, where Φ_{i1} contains single output operations and Φ_{i2} contains empty output operations. Let us also define the set

$\Phi_{i1}[p_{io}] = \{\phi_i \in \Phi_{i1} | F_i(p_{io}, \phi_i) \neq \emptyset\}$

(in other words the set $\Phi_{i1}[p_{io}]$ contains all $\phi_i \in \Phi_{i1}$ that are labels of arcs emerging from the initial state of the module $\mathcal{M}_i$).

Then we have the following definitions.

Definition 5.3.1.3.1.
Let $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a proper refinement set. Let i, j $\in$ {0, ..., n}, $\phi_i \in \Phi_{i1}$ and $\phi_j \in \Phi_{j1}$. 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 5.3.1.3.2.
Let $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a proper refinement set and let i, j $\in$ {0, ..., n}. Then $\mathcal{M}_i$ and $\mathcal{M}_j$ are said to be *output-distinguishable* if $\forall \phi_i \in \Phi_{i1}[p_{io}]$ and $\phi_j \in \Phi_{j1}[p_{jo}]$, ϕ_i and ϕ_j are fully output-distinguishable.

Definition 5.3.1.3.3.
Let $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i = 0,...,n}$ be a proper 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.

5.3.2. Fundamental test function and the refinement testing theorem.

As for stream X-machines, we define a fundamental test function of a refinement that transforms sequences from Φ^* into sequences from Σ^*.

Definition 5.3.2.1.
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).

· $t(1) = 1$,

· $t_M(1) = m_o$,

where m_o is the initial memory value of $\mathcal{M}$ and 1 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 $(m, y_i(m_i)) \in \text{dom}\, \phi_{k+1}$, 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 $t(\phi_1 ... \phi_{k+1})$.

1.b) Otherwise

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

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

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

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

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 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 5.3.2.2.

Let $\mathcal{M}'$ be the refinement from examples 5.1.6 and 5.3.1.2.1.1 (i.e. we consider the augmented version). Let name $\in$ STRINGS_n be a valid user name and psw1, psw2 $\in$ STRINGS_n be a valid and an invalid password, respectively, for this username with |name|, |psw1|, |psw2| ≥ 1. Then, we illustrate the construction of t and t_M with the following examples.

· t(`enter_name`) = name *`enter`*
· t_M(`enter_name`) = (*`in_acc`*, name)

· t(`enter_name good_psw`) = name *`enter`* psw1 *`enter`*
· t_M(`enter_name good_psw`) = (*`in_acc`*, name)

· t(enter_name wrong_pws1) = name *enter* psw2 *enter*
· t_M(enter_name wrong_psw1) = (*in_acc*, name)

· t(enter_name ignore_command) = name *enter* str *enter*,
where str $\in$ $STRINGS_n$, str $\neq$ ’exit’
· t_M(enter_name ignore_command) = $\emptyset$

· t(enter_name good_psw exit_system) = name *enter* psw1 *enter* ’exit’*enter*
· t_M(enter_name good_psw exit_system) = (*in_acc*, empty_seq)

· t(enter_name ignore_command exit_system) = name *enter* str *enter*
· t_M(enter_name ignore_command exit_system) = $\emptyset$

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

Theorem 5.3.2.3.
Let $\mathcal{M}$ = (I, Γ_2, Q, M, Φ, F, q_o, m_o), $\mathcal{M}$* = (I, Γ_2, Q*, M, Φ, 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 proper refinement set, $\mathcal{M}_i$ = (Σ, Γ_1, P_i, M_i, Φ_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 $\mathcal{M}$*’ the refinement of $\mathcal{M}$* w.r.t. Ref* and let f and f* the functions computed by $\mathcal{M}$’ and $\mathcal{M}$*’ respectively. We assume that $\mathcal{M}$*’ is a simple and complete refinement. Let also T and W be a transition cover and a characterisation set of $\mathcal{A}$, Z = Φ^k W $\cup$ Φ^{k-1} W $\cup$ ... $\cup$ W, with k a positive integer and t: Φ^* $\rightarrow$ Σ^* 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*.

Proof:
First, let us introducewing 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 5.3.2.3.1.

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 $\mathcal{M}$*' 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 $s_1...s_ks_{k+1} = t(\phi_1...\phi_k\phi_{k+1})$. If $f(s) = f^*(s)$, $\forall$ s $\in$ Pref($s_1...s_ks_{k+1}$), 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 $\mathcal{M}$*' 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 $\mathcal{M}$*' in the above mentioned states and memory values.

Let σ = head(s_{r+1}). Let γ be the output produced by $\mathcal{M}$' when it receives σ. 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 (see lemma 5.1.4). Hence $\gamma \in \Gamma_1$ (this is because the refinement set $\mathcal{R}$ is proper). From the hypothesis, it follows that $\mathcal{M}$*' produces the same output γ when σ is applied. Now, we have two possible cases: σ will cause $\mathcal{M}$*' to perform a type A or a type B transition. If the transition was of type B, then we would have $\gamma \in \Gamma_2$ (this is because the refinement set $\mathcal{R}$ is proper). 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 $\mathcal{M}$*' 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 $\mathcal{M}$*' 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 $\gamma_2 \in \Gamma_2$ (see lemma 5.1.5) . 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. $\square$

Then, from the lemma above it follows that q_0 and $q_0{}^*$ are TZ-equivalent as states in $\mathcal{A}$ and $\mathcal{A}^*$ respectively. From Theorem 4.1.4.1.6 (Chow), it follows that $\mathcal{A}$ and Min($\mathcal{A}^*$) are isomorphic, where $\mathcal{A}^*$ is the associated automaton of $\mathcal{M}^*$ (or alternatively that q_0 and $q_0{}^*$ are Φ^*-equivalent). Without loss of generality we assume that $\mathcal{A}^*$ is accessible. Then

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

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_0 and $q_0{}^*$ 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')≤ card(Q) + k, it follows that

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

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 $P = T(\Phi^k \cup \Phi^{k-1} \cup ... \cup \{1\})$ 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 $q_0{}^*$ to $q_j{}^*$. Since $q_i = g(q_j{}^*)$, there also exists a path $\phi_1...\phi_k$ from q_0 to q_i.

Since

$$f(s) = f^*(s), \forall\, s \in \text{Pref}(t(TZ)),$$

it follows that

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

If $(z_i, y_i, \mathcal{M}_i) = \text{Ref}(q_i)$ and $(z_j{}^*\ y_j{}^*, \mathcal{M}_j{}^*) = \text{Ref}(q_j{}^*)$, 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 ∈ {0, ..., n} and j ∈ {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 ∈ {0, ..., n'} and i ∈ {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 (this follows easily from definition 5.1.1). Hence, by applying proposition 3.4.1.1.7 and lemma 3.4.2.2, it follows that $\mathcal{M}'$ and $\mathcal{M}^{*\prime}$ compute the same function.
⑥

5.3.3. 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 proper and 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. $\Gamma_1 \cap \Gamma_2 = \emptyset$)
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.
8. The number of states of $\mathcal{M}^*$ is bounded by a certain number, say n'.

Then, under these circumstances Y = Pref(t(TZ)) is an *adequate* test set, 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 $\mathcal{A}$

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

· 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. We have shown that 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. 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 (i.e. completeness of the refinement, output-distinguishability of the ϕ's, etc.) can be easily translated into changes in the corresponding implementations. 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. 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.

Clearly, it is much more difficult to automate the process of generating the test set required by this method than the one required by the method presented in the previous chapter. This is because in this case the test function generates a sequence of input characters rather then a single character for each extra ϕ it processes. However, we could require that each such sequence of inputs has the length less then a certain number *l*; 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 5.3.1.2.1). If *l* is sufficiently small, it could be possible to generate the test set automatically (obviously, the complexity of the algorithm will depend on the complexity of the basic functions of the refinement modules and the control machine).

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.

5.4. Stream X-machine specification of a word processor.

We illustrate the concept of refinement with a specification of a simple word processor that allows the following operations:
· type a character;
· delete a character;
· move the cursor to the right or to the left (i.e. the inputs associated with these actions will be called *move_r* and *move_l* respectively);
· select (highlight) text; this can be done as follows:
the word processor starts selecting text when a certain input (called *sel_text*) is received
the text is selected by moving the cursor to the right or to the left
the action ends when a certain input (called *des_text*) is received, when a character is inserted or deleted or when the main menu of the processor is selected.

The word processor has a main menu with the following options: ’cut’, ’paste’, ’copy’, ’search’ and ’replace’. We assume that the ’cut’ and ’copy’ operations are active only when some text has been highlighted. We also assume that the ’paste’ operation is active only when there is some text copied into the clipboard.

We do not specify the user interface of the system in the sense that the inputs we shall be using will not match exactly the physical ones.

We shall assume that the main menu can be activated and deactivated by two inputs called *sel_menu* and *des_menu* respectively. There are also five inputs (i.e. *sel_copy*, *sel_paste*, *sel_cut*, *sel_search*, *sel_replace*) that activate each of the menu options.

When, the ’search’ option is chosen, the user will be required to enter the string to be searched for, the direction of search and to choose one of the following options:

· find the next string of characters that matches the string to be searched for; if the search is successful, the string found will be highlighted;

· end the search

The corresponding input to the word processor in this case will be a tuple (opt, str, dir), where opt represents one of the above options, str is the string to be searched for and dir is the direction of search.

When, the ’replace’ option is chosen, the user will be required to enter the string to be searched for, the replacement string and to choose one of the following options:

· find the next string of characters that matches the string to be searched for; the string found will be highlighted;

· replace the string that has been found (if any) and find the next string of characters that matches the string to be searched for; again, this will be highlighted;

· replace all the strings that match the string to be searched for; in this case no part of the document will be highlighted.

· end the replace operation.

Similar to the search operation, we shall consider that the system receives an input (opt, str1, str2), where opt is one of the above options, str1 is the string to be searched for and str2 the replacement string. Unlike the ’search’ operation there will be only one search direction, from the position of the cursor downwards.

Inputs and outputs

Let CHARACTERS be the set of characters that the system can process and let

CHARACTERS’ = CHARACTERS $\cup$ {*enter*}
STRINGS = CHARACTERS*,
STRINGS’ = CHARACTER’*
STRINGS’$^+$ = STRINGS’ - {empty_seq},

where empty_seq denotes the empty string.

Let also

DIRECTIONS = {*up*, *down*}.

Then, the set of inputs received by the word processor will be:

$\Sigma = \Sigma_O \cup \Sigma_{Se} \cup \Sigma_{Re}$,

where

Σ_O = CHARACTERS $\cup$ {*back_space*, *enter*, *move_l*, *move_r*, *sel_text*, *des_text*, *sel_menu*, *des_menu*, *sel_cut*, *sel_paste*, *sel_copy*, *sel_search*, *sel_replace*},

Σ_{Se} = {f_next_{Se}, $cancel_{Se}$} × STRINGS$^+$ × DIRECTIONS,

Σ_{Re} = {f_next_{Re}, *repl*, *repl_all*, $cancel_{Re}$} × STRINGS$^+$ × STRINGS, We shall consider that the outputs produced by the system will be:

Γ = DOCUMENT_DISPLAYS × {*Doc*, *Me*, *Re*, *Se*} × MESSAGES,
where
DOCUMENT_DISPLAYS = STRINGS’× STRINGS’× STRINGS’.
MESSAGES = {’Text not found’,*empty_msg*}.

The display of the document is considered to be a tuple of three sequences (a, c, b), where a and b are the non-highlighted parts of the text and c the highlighted one.

Doc, *Me*, *Re*, *Se* stand for ’document’, ’menu’, ’replace’ and ’search’ respectively and indicate which one of these is currently active.

The word processor will display the message ’Text not found’ when it performs an unsuccessful search operation.

5.4.1. Stream X-machine specification.

We specify the word processor in two stages. First, we shall construct a stream X-machine specification of the system without detailing the ’search’ and ’replace’ operations. This machine will later on be refined and the ’search’ and ’replace’ operations will be expanded using appropriate refinement modules.

The set of documents the word processor operates on is defined as
DOCUMENTS = STRINGS’ × STRINGS’ × STRINGS’ × POSITIONS,
where
POSITIONS = {*left*, *right*}.

The document is considered to be a tuple (x, z, y, p), where x is the first non-highlighted part of the document and rev(y) is the other non-highlighted part of the document. If p = *right*, then the selected text is z and the cursor is on the right hand side of this. If p = *left*, then the highlighted text is rev(z) and the cursor is on the left hand side of this.

For example, if the document is
’ab**cde**f g’
↑
then x = ’ab’, y = ’g f’, z = ’cde’ and p = *right* (i.e. ↑ marks the position of the cursor).
If the document is
’ab**cde**f g’
↑
then x = ’ab’, y = ’g f’, z = ’edc’ and p = *left*.

If no part of the document is selected (i.e. z = empty_seq), then the value of p is not relevant.

The position of the cursor with respect to the selected text will influence the 'select text' operation. For example, let us imagine that we are selecting text by moving the cursor one position to the right. If the cursor is positioned on the right of the highlighted text, then the first character to the right of the cursor will be also highlighted. If the cursor is positioned on the left of the highlighted text, then the leftmost character of the highlighted text will be deselected.

5.4.2. The control machine.

The control stream X-machine $\mathcal{M}$ will be defined as follows.

1. The input set is

$$I = \Sigma_0 \cup \text{DOCUMENTS}.$$

Since $\mathcal{M}$ will not specify in detail the 'search' and 'replace' operations, the inputs associated with these operations (Σ_{Se} and Σ_{Re}) are not included in I. Instead, if a 'search' or 'replace' operation has been performed, the machine will receive an input representing the updated document and will simply replace the current document with this new one. The way in which the updated document is obtained will be specified later by an appropriate refinement module.

2. The output set is

$$\Gamma_2 = \text{DOCUMENT\_DISPLAYS} \times \{\mathtt{Doc}, \mathtt{Me}, \mathtt{Re}, \mathtt{Se}\} \times \{\mathtt{empty\_msg}\}.$$

3. The state set is

Q = {Typing, SelectingText, Menu, Search, Replace).

The initial state is Typing.

4. The memory is

M = DOCUMENTS × STRINGS'

The current memory value will be denoted by ((x, y, z, p), c), where (x, y, z, p) represents the current document and c is the text copied into the clipboard.

The initial memory value is

m_0 = ((empty_seq, empty_seq, empty_seq, `left`), empty_seq).

5. The type is

Φ = {`type`, `delete`, `move`, `select_text`, `activate_select_text`, `deactivate_select_text`, `select_menu`, `deselect_menu`, `cut`, `paste`, `copy`, `select_search`, `select_replace`, `update_document`}.

6. The transition diagram is represented in figure 5.10.

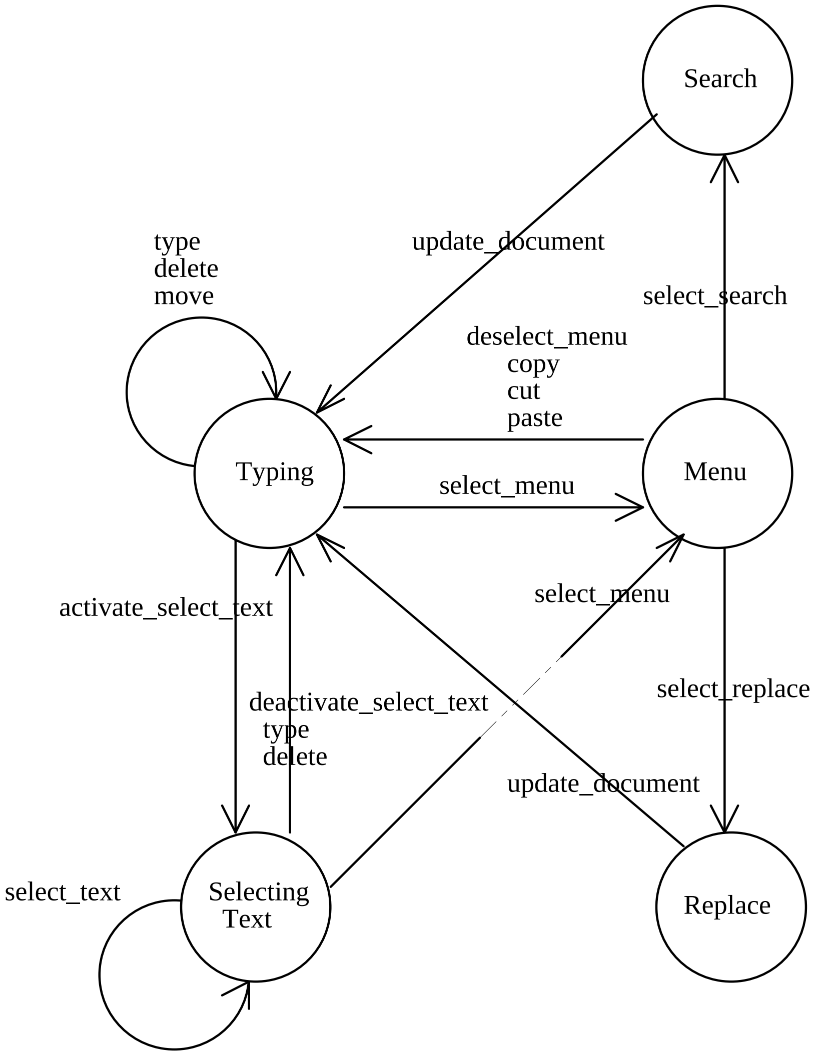


Figure 5.10.

7. Basic function definitions.

Before we give the formal definitions of the processing functions we define the following auxiliary functions.

· `display_doc`: DOCUMENTS → DOCUMENT_DISPLAYS

$\lceil$ $(x, z, \mathrm{rev}(y))$ if $p = \mathtt{right}$

$$\texttt{display\_doc}(x, z, y, p) = \begin{cases} \\ (x, \mathrm{rev}(z), \mathrm{rev}(y)), \text{ if } p = \texttt{left} \end{cases}$$

i.e. it displays a document.

· `mirror`: DOCUMENTS → DOCUMENTS

$$\texttt{mirror}(x, z, y, p) = \begin{cases} (y, z, x, \texttt{left}), \text{ if } p = \texttt{right} \\ (y, z, x, \texttt{right}), \text{ if } p = \texttt{left} \end{cases}$$

i.e. it reverses a document.

· `move_left_doc`: DOCUMENTS → DOCUMENTS

$\texttt{move\_left\_doc}(x, y, z, p) = (x', y', z', p)$,

where

$$x' = \begin{cases} \mathrm{front}(x), \text{ if } z = \text{empty\_seq} \\ x, \text{ if } z \in \text{STRINGS}'^{+} \end{cases}$$

$$y' = \begin{cases} y\ \mathrm{rear}(x)\ , \text{ if } z = \text{empty\_seq} \\ y\ \mathrm{rev}(z), \text{ if } z \in \text{STRINGS}'^{+} \text{ and } p = \texttt{right} \\ \text{yz, if } z \in \text{STRINGS}'^{+} \text{ and } p = \texttt{left} \end{cases}$$

z' = empty_seq.

i.e. it moves the cursor one position to the left.
If no part of the document is selected (i.e. z = empty_seq), then:

if the cursor is not on the left of the document, then the cursor moves one position to the left;

otherwise, the document is left unchanged.

If a part of the document is selected (i.e. $z \neq$ empty_seq), then this is deselected (i.e. z' = empty_seq) and the cursor is positioned on the left of the part of the text that had been selected.

For example:

’abcdefg’ is transformed into ’abcdefg’,
↑ ↑

’abcdefg’ remains unchanged,
↑

’ab**cd**efg’ is transformed into ’abcdefg’.

↑ ↑

· `select_left_doc`: DOCUMENTS → DOCUMENTS

`select_left_doc`(x, y, z, p) = (x', y', z', p),

where

$$x' = \begin{cases} x, \text{ if } z \in \text{STRINGS}^{*+} \text{ and } p = \texttt{right} \\ \text{front}(x), \text{ if } z = \text{empty\_seq or } p = \texttt{left} \end{cases}$$

$$y' = \begin{cases} y \text{ rear}(z), \text{ if } p = \texttt{right} \\ y, \text{ if } p = \texttt{left} \end{cases}$$

$$z' = \begin{cases} \text{front}(z), \text{ if } z \in \text{STRINGS}^{*+} \text{ and } p = \texttt{right} \\ z \text{ rear}(x), \text{ if } z = \text{empty\_seq or } p = \texttt{left} \end{cases}$$

$$p' = \begin{cases} p, \text{ if } z \in \text{STRINGS}^{*+} \\ \texttt{left}, \text{ if } z = \text{empty\_seq} \end{cases}$$

i.e. it selects text by moving the cursor one position to the left.

If the cursor is on the left of the document, then the document remains unchanged. Otherwise

if no part of the document is selected (z = empty_seq), then the first character on the left hand side of the cursor is selected and the cursor moves one position to the left;

if a part of the document is selected ($z \neq$ empty_seq), then

if the cursor is on the right hand side of the selected text, then the first character on the left hand side of the cursor is deselected and the cursor moves one position to the left;

if the cursor is on the left hand side of the selected text, then the first character on the left hand side of the document is selected and the cursor moves one position to the left.

For example:

’abcdefg’ is transformed into ’ab**d**efg’,
↑ ↑

’abcdefg’ remains unchanged,
↑

’ab**cd**efg’ is transformed into ’ab**c**defg’.
↑ ↑

Then the basic processing functions are defined as follows:

· dom $\texttt{type} = M \times \text{CHARACTERS}'$

$\texttt{type}(((x, z, y, p), c), ch) = (\gamma, ((x', z', y, p), c))$

where

$\gamma = (\texttt{display\_doc}(x', z', y, p), \textit{Doc}, \textit{empty\_msg}),$
$x' = x\ ch,$
$z' = \text{empty\_seq}$

i.e. if no part of the document is selected, then $ch \in$ CHARACTERS' is inserted on the left of the cursor. Otherwise, the selected text is removed and *ch* is inserted on the left hand side of the cursor.

· dom $\texttt{delete} = M \times \{\textit{back\_space}\}$

$\texttt{delete}(((x, z, y, p), c), \textit{back\_space}) = (\gamma, ((x', z', y, p), c))$

where

$\gamma = (\texttt{display\_doc}(x', z', y, p), \textit{Doc}, \textit{empty\_msg})$

$$x' = \begin{cases} x, \text{ if } z \in \text{STRINGS}'^{+} \\ \text{front}(x), \text{ if } z = \text{empty\_seq} \end{cases}$$

$z' = \text{empty\_seq}$

i.e. it deletes a character.
If no part of the document is selected (i.e. z = empty_seq), then
if the cursor is not on the left hand side of the document, then the first character on the left hand side of the cursor is deleted;
otherwise the document remains unchanged.
If a part of the document is selected (i.e. $z \neq$ empty_seq), then this is removed.

· dom $\texttt{move} = M \times \{\textit{move\_l}, \textit{move\_r}\}$

$\texttt{move}(((x, z, y, p), c), mv) = (\gamma, ((x', z', y', p'), c))$

where

$\gamma = (\texttt{display\_doc}(x', z', y', p'), \textit{Doc}, \textit{empty\_msg})$

$$(x', y', z', p') = \begin{cases} \texttt{move\_left\_doc}(x, z, y, p), \text{ if } mv = \textit{move\_l} \\ \texttt{mirror}(\texttt{move\_left\_doc}(\texttt{mirror}(x, z, y, p))), \\ \qquad \text{if } mv = \textit{move\_r} \end{cases}$$

i.e. when the system is in the state Typing, the cursor is moved to the left or to the right when $\texttt{move\_r}$ or $\texttt{move\_l}$ are received.

· dom $\texttt{select\_text} = \mathrm{M} \times \{\texttt{move\_l}, \texttt{move\_r}\}$

$\texttt{select\_text}(((x, z, y, p), c), mv) = (\gamma, ((x', z', y', p'), c))$

where

$\gamma = (\texttt{display\_doc}(x', z', y', p'), \texttt{Doc}, \texttt{empty\_msg})$

$$(x', z', y' p') = \begin{cases} \texttt{select\_left\_doc}(x, y, z, p), \text{ if } mv = \texttt{move\_l} \\ \texttt{mirror}(\texttt{select\_left\_doc}(\texttt{mirror}(x, y, z, p))), \\ \qquad\qquad\qquad\qquad \text{if } mv = \texttt{move\_r} \end{cases}$$

i.e. when the system is in the state SelectingText state, it selects text by moving the cursor to the left or to the right (i.e. when $\texttt{move\_r}$ or $\texttt{move\_l}$ are received).

· dom $\texttt{activate\_select\_text} = \mathrm{M} \times \{\texttt{sel\_text}\}$

activate_text$(((x, z, y, p), c), \texttt{sel_text}\} = (\gamma, ((x, z, y, p), c))$

where

$\gamma = (\texttt{display\_doc}(x, z, y, p), \texttt{Doc}, \texttt{empty\_msg})$

i.e. the word processor starts selecting text when $\texttt{sel\_text}$ is received.

· dom $\texttt{deactivate\_select\_text} = \mathrm{M} \times \{\texttt{des\_text}\}$

deactivate_text$(((x, z, y, p), c), \texttt{des_text}\} = (\gamma, ((x, z, y, p), c))$

where

$\gamma = (\texttt{display\_doc}(x, z, y, p), \texttt{Doc}, \texttt{empty\_msg})$

i.e. the word processor stops selecting text when $\texttt{des\_text}$ is received.

· dom $\texttt{select\_menu} = \mathrm{M} \times \{\texttt{sel\_menu}\}$

$\texttt{select\_menu}(((x, z, y, p), c), \texttt{sel\_menu}) = (\gamma, ((x, z, y, p), c))$

where

$\gamma = (\texttt{display\_doc}(x, z, y, p), \texttt{Me}, \texttt{empty\_msg})$

i.e. the main menu is selected using $\mathtt{sel\_menu}$.

· dom $\mathtt{deselect\_menu} = M \times \{\mathtt{des\_menu}\}$

$\mathtt{deselect\_menu}(((x, z, y, p), c), \mathtt{des\_menu}) = (\gamma, ((x, z, y, p), c))$

where

$\gamma = (\mathtt{display\_doc}(x, z, y, p), \mathtt{Doc}, \mathtt{empty\_msg})$

i.e. the main menu is deselected using $\mathtt{des\_menu}$.

· dom $\mathtt{cut} = \{((x, z, y, p), c) \in M \mid z \neq \text{empty\_seq}\} \times \{\mathtt{sel\_cut}\}$

$\mathtt{cut}(((x, z, y, p), c), \mathtt{sel\_cut}) = (\gamma, ((x, z', y, p), c'))$

where

$\gamma = (\mathtt{display\_doc}(x, z', y, p), \mathtt{Doc}, \mathtt{empty\_msg})$

$z' = \text{empty\_seq}$

$$c' = \begin{cases} z, \text{ if } p = \mathtt{right} \\ \text{rev}(z), \text{ if } p = \mathtt{left} \end{cases}$$

i.e. the 'cut' option can be selected only if a part of the document has been selected. In this case, the selected text is removed and copied into the clipboard.

· dom $\mathtt{paste} = \{((x, z, y, p), c) \in M \mid c \neq \text{empty\_seq}\} \times \{\mathtt{sel\_paste}\}$

$\mathtt{paste}((((x, z, y, p), c), \mathtt{sel\_paste}) = (\gamma, ((x', z', y, p), c))$

where

$\gamma = (\mathtt{display\_doc}(x', z', y, p), \mathtt{Doc}, \mathtt{empty\_msg})$

$x' = \text{xc},$

$z' = \text{empty\_seq}$

i.e. the 'paste' option can be selected only if the clipboard is not empty. In this case, the selected part of the document is removed and the text in the clipboard is copied into the document on the left of the cursor.

· dom $\mathtt{copy} = \{((x, z, y, p), c) \in M \mid z \neq \text{empty\_seq}\} \times \{\mathtt{sel\_copy}\}$

$\mathtt{copy}(((x, z, y, p), c), \mathtt{sel\_copy}) = (\gamma, ((x, z, y, p), c'))$

where

$$\gamma = (\texttt{display\_doc}(x, z, y, p), \mathit{Doc}, \mathit{empty\_msg})$$

$$c' = \begin{cases} z, \text{ if } p = \mathit{right} \\ \text{rev}(z), \text{ if } p = \mathit{left} \end{cases}$$

i.e. the 'copy' option can be selected if a part of the document has been selected. In this case, the selected text is copied into the clipboard.

· dom $\texttt{select\_search}$ M × $\{\mathit{sel\_search}\}$

$\texttt{select\_search}(((x, z, y, p), c), \mathit{des\_menu}) = (\gamma, ((x, z, y, p), c))$

where

$$\gamma = (\texttt{display\_doc}(x, z, y, p), \mathit{Doc}, \mathit{empty\_msg})$$

i.e. the 'search' option is selected using$\mathit{sel_search}$.

· dom $\texttt{select\_replace}$ M × $\{\mathit{sel\_replace}\}$

$\texttt{select\_replace}(((x, z, y, p), c), \mathit{des\_menu}) = (\gamma, ((x, z, y, p), c))$

where

$$\gamma = (\texttt{display\_doc}(x, z, y, p), \mathit{Doc}, \mathit{empty\_msg})$$

i.e. the 'replace' option is selected using$\mathit{sel_replace}$.

· dom $\texttt{update\_document}$ = M × DOCUMENTS

$\texttt{update\_document}(((x, z, y, p), c), (n\_x, n\_z, n\_y, n\_p)) =$
$(\gamma, ((n\_x, n\_z, n\_y, n\_p), c))$

where

$$\gamma = (\texttt{display\_doc}(n\_x, n\_z, n\_y, n\_p), \mathit{Doc}, \mathit{empty\_msg})$$

i.e. this function is used to perform the unrefined 'search' and 'replace' operations. The updated document (n_x, n_z, n_y, n_p) is received by the machine and replaces the current document.

5.4.3. Refinement.

We now detail the 'search' and 'replace' operations. This will be done using the operation of refinement. The refinement set will be $\mathcal{R} = \{(z_i, y_i, \mathcal{M}_i)\}_{i=0,..2}$ and the refinement function Ref: $Q \rightarrow \mathcal{R}$ will be defined by:

Ref(Typing) = Ref(SelectingText) = Ref(Menu) = $(z_0, y_0, \mathcal{M}_0)$,
Ref(Search) = $(z_1, y_1, \mathcal{M}_1)$,
Ref(Replace) = $(z_2, y_2, \mathcal{M}_2)$.

Since the only operations that need refining are the 'search' and 'replace' operations, the module $\mathcal{M}_0$ will only be used to read the appropriate inputs and pass them to $\mathcal{M}$. $\mathcal{M}_1$ and $\mathcal{M}_2$ will be used to detail the 'search' and 'replace' operations respectively.

The input alphabet of $\mathcal{M}_0$, $\mathcal{M}_1$ and $\mathcal{M}_2$ will be Σ. The output alphabet will be

Γ_2 = DOCUMENT_DISPLAYS × {*Se*, *Re*} × MESSAGES

(i.e. the output alphabet for $\mathcal{M}_1$ will be Γ_{21} = DOCUMENT_DISPLAYS × {*Se*} × MESSAGES; the output alphabet for $\mathcal{M}_1$ will be Γ_{22} = DOCUMENT_DISPLAYS × {*Re*} × MESSAGES; the output alphabet for $\mathcal{M}_0$ is not relevant since $\mathcal{M}_0$ will not produce any outputs; hence $\Gamma_2 = \Gamma_{21} \cup \Gamma_{22}$}

The refinement modules and the transfer functions are defined as follows.

5.4.3.1. The module $\mathcal{M}_0$.

1. The state set is $P_0 = \{p_{0.0}, p_{0.1}\}$; $p_{0.0}$ is the initial state and $p_{0.1}$ is the final state.
2. The memory set is $M_0 = \Sigma_0$.
3. The type is $\Phi_0 = \{\phi_0\}$, where
   dom $\phi_0 = \Sigma_0 \times \Sigma_0$,
   $\phi_0(\sigma, \sigma') = (\text{empty\_seq}, \sigma') \forall\, \sigma, \sigma' \in \Sigma_0$.
4. The next state transition diagram is represented in figure 5.11.

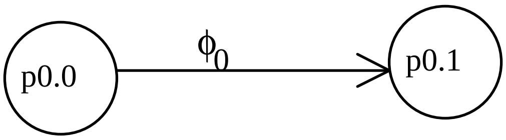


Figure 5.11.

Basically $\mathcal{M}_0$ reads all the inputs in Σ_0 and rejects the inputs in $\Sigma - \Sigma_0$ without producing any output.

5.4.3.2. The transfer functions z_0 and y_0.

The transfer functions

z_O: DOCUMENTS × STRINGS' → Σ_O

and

y_O: $\Sigma_O \to (\Sigma_O \cup \text{DOCUMENTS})$

are defined by:

$z_O((x, z, y, p), c) = \sigma_O$, where $\sigma_O \in \Sigma_O$ is an arbitrary fixed element.

$y_O(\sigma) = \sigma$.

5.4.3.3. Preparatory definitions.

Before continuing we define the following (partial) functions that we shall be needing for the definitions of the two remaining modules.

· `selected_text`: STRINGS'× POSITIONS → STRINGS (function)

$$\texttt{selected\_text}(z, p) = \begin{cases} z, \text{ if } p = \texttt{right} \\ \text{rev}(p), \text{ if } p = \texttt{left} \end{cases}$$

i.e. it outputs the part of the document that is selected.

· `found`: STRINGS'× STRINGS'^{+} → B (function)

[**Note**: B is the set of Booleans]

`found`$(x, f) = (\exists\, a, b \in$ STRINGS' such that$x = a\, f\, b)$.

i.e. `found` returns true if the string *x* contains the string *f*.

· `find_text`: STRINGS' × STRINGS'^{+} → STRINGS' × STRINGS' (partial function)

dom `find_text` = $\{(x, f) \in$ STRINGS'× $\text{STRINGS'}^{+} |$ `found`$(x, f)\}$

`find_text`$(x, f) = (a, b)$, where *a* and *b* are chosen such that

$x = a\, f\, b$ and

$\forall\, a' \in \{c \in \text{STRINGS'} | \exists\, \text{d} \in$ STRINGS' such that$x = c\, f\, \text{d}\}, |a| \leq |a'|$.

i.e. basically, `find_text` returns two strings *a* and *b* such that $x = a\, f\, b$ and *a* is the shortest string which satisfies this condition.

· `update_all`: STRINGS' × STRINGS' × STRINGS'^{+} × STRINGS' → STRINGS'× STRINGS' (function)

$\texttt{update\_all}(x, y, f, r) = (x', y')$, where

  if $\neg\texttt{found}(y, f)$, then $x' = x$ and $y' = y$

  otherwise, $(x', y') = \texttt{update\_all}(x\ a\ r, b, f, r)$, with $(a, b) =$ find_string(y, f)

i.e. if x and y are two strings of characters, then $\texttt{update\_all}(x, y, f, r) = (x', y')$, where $x'\,y'$ is *a* string obtained by replacing all the occurrences of f in $x\ y$ with r starting from the left most character of y. Also y' is the part of y that remains unchanged

5.4.3.4. The module $\mathscr{m}_1$.

1. The state set is

  $P_1 = \{p_{1.o}, p_{1.1}\}$;

$p_{1.o}$ is the initial state and $p_{1.1}$ is the final state.

2. The memory set is

  M_1 = DOCUMENTS.

3. The type is

  $\Phi_1 = \Phi_{11} \cup \Phi_{12}$,

where

  $\Phi_{11} = \{\texttt{find\_succ}_1, \texttt{find\_unsucc}_1\}$,

  $\Phi_{12} = \{\ \texttt{cancel\_search}\}$

4. The transition diagram is represented in figure 5.12.

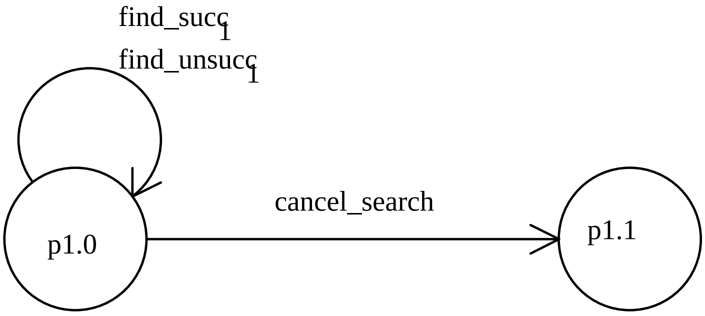


Figure 5.12.

5. The processing functions are defined as follows.

· dom $\texttt{find\_succ}_1 = \{((x, z, y, p), (\textit{f\_next}_{Se}, f, \mathrm{d}))\mid (x, z, y, p) \in$ DOCUMENTS, $f \in$ STRINGS$^+$, $\mathrm{d} \in$ DIRECTIONS and ((d = *down* and $\texttt{found}(\mathrm{rev}(y), f)$) or (d = *up* and $\texttt{found}(\mathrm{rev}(x), \mathrm{rev}(f))$))}

$\texttt{find\_succ}_1((x, z, y, p), (\textit{f\_next}_{Se}, f, \mathrm{d})) = (\gamma, (x', z', y', p'))$

where

  $\gamma = (\texttt{display\_doc}(x', z', y', p'), \textit{Se},$ empty_message) and

  if d = *down* then

$x' = x\ \texttt{selected\_text}(z, p)\ a$
$z' = f$
$y' = \text{rev}(b)$
$p' = \texttt{right}$
where a and b satisfy $(a, b) = \texttt{find\_text}(\text{rev}(y), f)$

if d = $\texttt{up}$ then
$x' = \text{rev}(b)$
$z' = f$
$y' = y\ a$
$p' = \texttt{right}$
where a and b satisfy $(a, b) = \texttt{find\_text}(\text{rev}(x), \text{rev}(f))$

i.e. it finds the next occurrence of the string f in the document. There are two possible search directions: ’down’ and ’up’.

· dom $\texttt{find\_unsucc}_1 = \{((x, z, y, p), (\texttt{f\_next}_{Se}, f, \text{d}))|\ (x, z, y, p) \in$ DOCUMENTS, $f \in \text{STRINGS}^+$, d $\in$ DIRECTIONS and $\neg$((d = $\texttt{down}$ and $\texttt{found}(\text{rev}(y), f)$) or (d = $\texttt{up}$ and $\texttt{found}(\text{rev}(x), \text{rev}(f))$))}

$\texttt{find\_unsucc}_1((x, z, y, p), (\texttt{f\_next}_{Se}, f, \text{d})) = (\gamma, (x, z, y, p))$

where
$\gamma = (\texttt{display\_doc}(x, z, y, p), Se,$ 'Text not found')

i.e. if the search is unsuccessful, then the document remains unchanged.

· dom $\texttt{cancel\_search} = \{((x, z, y, p), (\texttt{cancel}_{Se}, f, \text{d}))|\ (x, z, y, p) \in$ DOCUMENTS, $f \in$ STRINGS, d $\in$ DIRECTIONS}

$\texttt{cancel\_search}\ ((x, z, y, p), (\texttt{cancel}_{Se}, f, \text{d}))\ (\gamma, (x, z, y, p))$

where
γ = empty_seq

i.e. if the 'cancel' option is chosen, then the document is left unchanged.

5.4.3.5. The transfer functions z_1 and y_1.

The transfer functions
z_1: DOCUMENTS × STRINGS' → DOCUMENTS
and
y_1: DOCUMENTS → ($\Sigma_o \cup$ DOCUMENTS)
are defined by:
$z_1((x, z, y, p), c) = (x, z, y, p)$

$y_1(x, z, y, p) = (x, z, y, p)$.

5.4.3.6. The module $\mathscr{m}_2$.

1. The state set is

$P_2 = \{p_{2.0}, p_{2.1}, p_{2.2}\}$;

$p_{2.0}$ is the initial state and $p_{2.2}$ is the final state.
$\mathscr{m}_2$ will be in the state $p_{2.1}$ when some test has already been found and selected; otherwise $\mathscr{m}_2$ will be in the state $p_{2.0}$.

2. The memory set is

M_2 = DOCUMENTS.

3. The type is

$\Phi_2 = \Phi_{21} \cup \Phi_{22}$,

where

Φ_{21} = {`find_succ`$_2$, `find_unsucc`$_2$, `replace&find_succ`, `replace&find_unsucc`, `replace_all_succ`, `replace_all_unsucc`}.

Φ_{22} = {`cancel_replace`}

4. The transition diagram is represented in figure 5.13.

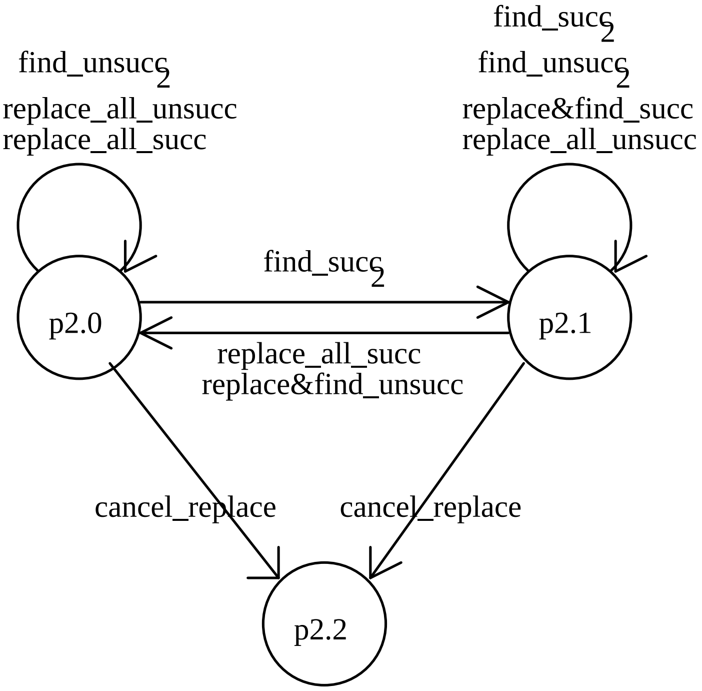


Figure 5.13.

5. The processing functions are defined as follows.

· dom `find_succ`$_2$ = {((*x*, *z*, *y*, *p*), (*`f_next`*$_{Se}$, *f*, *r*))| (*x*, *z*, *y*, *p*) ∈ DOCUMENTS, *f* ∈ STRINGS$^+$, *r* ∈ STRINGS and `found`(rev(*y*), *f*)}

`find_succ`$_2$((*x*, *z*, *y*, *p*), (*`f_next`*$_{Se}$, *f*, *r*)) = (γ, (*x*’,*z*’,*y*’,*p*’))

where

γ = (display_doc(x',z',y',p'),*Se*, empty_message)
x' =x selected_text(z, p) a
z' =f
y' = rev(b)
p' =*right*
where a and b satisfy (a, b) = find_text(rev(y), f)

i.e. this is similar to $\texttt{find\_succ}_1$ when the direction of search is ’down’.

· dom $\texttt{find\_unsucc}_2$ = {((x, z, y, p), (f_next_{Se}, f, r))| (x, z, y, p) $\in$ DOCUMENTS, $f \in$ STRINGS$^+$, $r \in$ STRINGS and ¬(found(rev(y), f))}

$\texttt{find\_unsucc}_2$((x, z, y, p), (f_next_{Se}, f, r)) = (γ, (x, z, y, p))

where

γ = (display_doc(x, z, y, p), *Se*, 'Text not found')

i.e. this is similar to $\texttt{find\_unsucc}_1$ when the direction of search is 'down'.

· dom replace&find_succ = {((x, z, y, p), (*repl*, f, r))| (x, z, y, p) $\in$ DOCUMENTS, $f \in$ STRINGS$^+$, $r \in$ STRINGS and found(rev(y), f)}

replace&find_succ((x, z, y, p), (*repl*, f, r)) = (γ, (x', z', y', p'))

where

γ = (display_doc(x', z', y', p'), *Re*, empty_message}

x' = x r a
z' = f
y' = rev(b)
p' = *right*
here a and b satisfy (a, b) = find_text(rev(y), f)

i.e. it replaces the selected text with r, deselects it and finds the next occurrence of f in the document on the right hand side of the text that had been selected. The text found is selected and the cursor is positioned on its right hand side.

· dom replace&find_unsucc = {((x, z, y, p), (*repl*, f, r))| (x, z, y, p) $\in$ DOCUMENTS, $f \in$ STRINGS$^+$, $r \in$ STRINGS and found(rev(y), f)}

replace&find_unsucc((x, z, y, p), (*repl*, f, r)) = (γ, (x', z', y, p))

where

γ = (display_doc(*x*’,*z*’,*y*, *p*), *Re*, ’Text not found’}

x’ =*x r*

z’ = empty_seq

i.e. this function is applied when there is no occurrence of *f* in the document on the right hand side of the selected text. In this case, the selected text is replaced by *r* and deselected and the cursor is positioned on the right hand side of this.

· dom replace_all_succ = {((*x*, *z*, *y*, *p*), (*repl*, *f*, *r*))| (*x*, *z*, *y*, *p*) $\in$ DOCUMENTS, $f \in$ STRINGS$^+$, $r \in$ STRINGS and (*f* = selected_text(*z*, *p*) or found(rev(*y*), *f*))}

replace_all_succ((*x*, *z*, *y*, *p*)), (*repl_all*, *f*, *r*)) = (γ, (*x*’,*z*’,*y*’,*p*))

where

γ = ((display_doc(*x*’,*z*’,*y*’,*p*), *Re*), empty_message)

$$x' = \begin{cases} x\ r\ a, \text{ if } \texttt{selected\_text}(z, p) = f \\ x\ \texttt{selected\_text}(z, p)\ a, \text{ otherwise} \end{cases}$$

z’ = empty_seq

y’ = rev(*b*)

(*a*, *b*) = update_all(empty_seq, rev(*y*), *f*, *r*)

i.e. this function is applied only if the selected text is identical to *f* or there is at least one occurrence of *f* on the left hand side of the selected text. The operations performed by the function on the document are:

if the selected text is identical to *f*, then it is replaced with *r*

and/or

replace all the occurrences of *f* in the left part of the document with *r*.

No part of the resulting document will be selected.

· dom replace_all_unsucc = {((*x*, *z*, *y*, *p*), (*repl*, *f*, *r*))| (*x*, *z*, *y*, *p*) $\in$ DOCUMENTS, $f \in$ STRINGS$^+$, $r \in$ STRINGS and ¬(*f* = selected_text(*z*, *p*)) and ¬(found(rev(*y*), *f*)))}

replace_all_unsucc((*x*, *z*, *y*, *p*)), (*repl_all*, *f*, *r*)) = (γ, (*x*, *z*, *y*, *p*))

where

γ = ((display_doc(*x*', *z*', *y*', *p*), *Re*), 'Text not found').

i.e. if the selected text is not identical to *f* and there are no occurrences of *f* on the left hand side of the selected text, then the document is left unchanged and the word processor gives an appropriate message.

· dom `cancel_replace` = $\{((x, z, y, p), (\mathtt{cancel}_{\mathtt{Re}}, f, r)) | \ (x, z, y, p) \in$ DOCUMENTS, $f \in$ STRINGS, d $\in$ STRINGS$\}$

`cancel_replace`$((x, z, y, p), (\mathtt{cancel}_{\mathtt{Re}}, f, r)) =$
$(\gamma, (x, z, y, p)),$
where
γ = empty_seq;

i.e. if the ’cancel’ option is chosen, then the document is left unchanged.

5.4.3.7. The transfer functions z_2 and y_2.

The transfer functions are $z_2 = z_1$ and $y_2 = y_1$.

5.4.4. Further refinements.

The specification we have obtained can be refined further. For example, we can detail the way in which the inputs associated with the search and replace operations are entered. This results in a new level of refinement which specifies how the string to be searched for, the replacement string and the direction of search can be obtained from simple characters and possibly some extra inputs (i.e. we need to consider how to mark the beginning and the end of these strings - this can be done using the `enter` key or possibly some new inputs). This will be similar to example 5.1.6.

5.4.5. Using stream X-machines (X-modules) for specifying the processing functions.

The processing functions of $\mathcal{M}$ and $\mathcal{M}_0$ are straightforward, involving simple operations on strings like concatenation, removing the first of the last element of the string, etc. The processing functions of $\mathcal{M}_1$ and $\mathcal{M}_2$ are more complex. However, they can be constructed easily from three functions (`found`, `find_text`, `update_all`) that we have defined recursively in section 5.4.3.3 In turn, these can be specified easily using quite simple stream X-machines (X-modules).

For example, let us consider `update_all`. This is not itself a stream function, but it can be defined using $\mathcal{M}_u$, the following stream X-module.

1. The input alphabet is CHARACTERS’.
2. The output alphabet is STRINGS’× STRINGS’.
3. The state set is Q = {q}.
4. The memory set is
   M = STRINGS’× STRINGS’× STRINGS’× STRINGS’$^+$ × STRINGS’.
5. The type is
   Φ = {read_input1, read_input2, read&update}
6. The transition diagram is represented in figure 5.14.

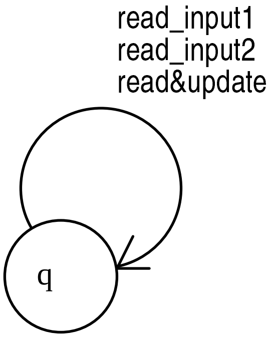


Figure 5.14.

7. The processing functions are defined as follows.

· dom read_input1 = {((*a*, *b*, *c*, *d*, *e*), *ch*)| (*a*, *b*, *c*, *d*, *e*) ∈ M, *ch* ∈ CHARACTERS’ and |*c*| < |*d*|-1}

read_input1((*a*, *b*, *c*, *d*, *e*), *ch*) = (γ, (*a*, *b*, *c*’,*d*, *e*))

where

γ = (*a*, *b c*’),*c*’ =*c ch*.

· dom read_input2 = {((*a*, *b*, *c*, *d*, *e*), *ch*)| (*a*, *b*, *c*, *d*, *e*) ∈ M, *ch* ∈ CHARACTERS’, |*c*| ≥ |*d*|-1 and *c ch* ≠ *d*}

read_input2((*a*, *b*, *c*, *d*, *e*), *ch*) = (γ, (*a*, *b*’,*c*’,*d*, *e*))

where

γ = (*a*, *b*’ *c*’),*b*’ =*b* head(*c*), *c*’ = tail(*c*) *ch*.

· dom read&update = {((*a*, *b*, *c*, *d*, *e*), *ch*)| (*a*, *b*, *c*, *d*, *e*) ∈ M, *ch* ∈ CHARACTERS’, *c ch* = *d*}

read&update((*a*, *b*, *c*, *d*, *e*), *ch*) = (γ, (*a*’,*b*’,*c*’,*d*, *e*)

where

$\gamma = (a', b' c')$, where $a' = a\ b\ e$, b' = empty_seq, c' = empty_seq.

Now, let m_0 = (x, empty_seq, empty_seq, f, r) the initial memory value, (i.e. f is the string to be searched for and r is the replacement string). Let y be the string of inputs received by $\mathcal{M}_u$, let g be the corresponding output sequence and let γ = rear(g) be the last symbol of the sequence g. Then $\gamma = (z, w)$, where $z\ w$ is a string obtained by replacing all the occurrences of f in $x\ y$ by r starting from the leftmost character of y. Also w is the part of y that remains unchanged. Therefore

`update_all`(x, y, f, r) = (z, w).

Let

f_u: (STRINGS' × STRINGS' × STRINGS' × STRINGS'$^+$ × STRINGS') × STRINGS' → (STRINGS'× STRINGS')*

be the function computed by $\mathcal{M}_u$. Then

`update_all`(x, y, f, r) = rear(f_u((x, empty_seq, empty_seq, f, r), y)).