19/3/98

PART 2 THEORETICAL FOUNDATIONS

Chapter 6.

The Theory of X-Machines.

Summary. Basic theory of X-machines, Stream X-machines and stream functions. Functional Refinement. State refinement. The fundamental theorem of refinement.

The first part of this book concentrated on practical methods for designing a comprehensive formal model of a system in an integrated way that related naturally to the initial stages of requirements identification and capture and allowed for the generation of complete functional test sets. It was formulated on the development of models through a variety of levels of functionality and detail. In this chapter we look at some of the theory that underpins this approach. The philosophy is essentially one of the algebraic modelling of computation. This requires some previous knowledge of the theory of finite state machines [23] and of Turing machines [22], typically material taught in undergraduate computer science degree courses. The chapter begins by considering the formal definition of an X-machine and some of the basic terminology. In section 2 we look at stream X-machines, the key model used in part 1. The last section introduces the idea of machine refinement - the mechanism for developing a stream X-machine model into more complex models with extended functionality.

6.1. The basic theory of X-machines.

The Turing model [22], has been a cornerstone of the theory of computation for many years and the Chomsky hierarchy of machines a useful mechanism for categorizing machines and languages of different capabilities. Despite their theoretical value, these computational models are not used by software engineers, the principle reason being that they are not readily amenable to traditional analysis and system development. Indeed, the data structures these machines use and process is much too restrictive and low level to be used in practical situations - in most cases such data structures are stacks or tapes and the operation allowed on these are fairly primitive; for example, not even the addition of two integers is straightforward to describe using such low level models. Also, in practice, one would like to have the freedom to be able to choose the system data and the way this it is processed that best suits the application in hand - as long as these can be implemented on a computer, of course.

One could argue that the development of any software system is a hierarchical process in which each level will use things that result from the previous levels. The depth of this process will depend on the tools that the current application will use; in most cases the bottom level will coincide with methods (functions, procedures, modules, etc.) from the libraries available to the implementation medium used.

Even though the existing computational models are of very little practical use, there is, however, a model, namely the X-machine, that easily accommodates this hierarchical approach and could therefore be of great practical value. The X-machine is not a computational model itself, it is rather a general framework that abstracts the common features of the main existing models (i.e. Finite State Machine, Pushdown Machine, Turing Machine and other standard types of machine) and can easily be adapted to suit the needs of many practical applications. The model was introduced in the first chapter and we will now provide a formal definition.

Basic concepts.

The full and most general definition of an X-machine is our starting point. Such a machine has a finite set of states, Q, together with some arrows that link some states with some others. Each arrow is labelled with a function taken from a set of functions Φ, (these functions are possibly partial functions). Each of these functions operates on a given data set, X. We identify a set, I, of initial states, so that the machine is always started off in one of these states, usually there is a unique start state. There is also specified a set, T, of terminal or final states. In many cases this is the set of all states, in certain situations, however, we may wish to restrict what states can be terminal states and then we will be interested in the conditions under which the machine halts in such a state.

The main issue left is how such a machine would operate. it would need some stimulatuion and this will by applying a series of inputs to it. The response of the system will be to create some visible output behaviour. So we postulate the existence of a set *Input*, of all possible inputs to the machine and a set, *Output* of all possible outputs. The way these inputs are applied to the mechine is by an input function α, which takes an input and interprets it as an element of X. Having supplied the machine with a starting value of X and an initial starting state it will try to start computing by searching for an arrow leaving that state which is labelled by a function ϕ from Φ which can process the initial value x of X. Note that there may be none, one or many such functions such that the value x lies within their domain. One of these functions (if any exist) is selected randomly, say ϕ and the new value $\phi(x)$ of X calculated, at the same time the machine moves to the state pointed to by the arrow labelled by ϕ. The process then continues until no further processing is possible. Then we may interpret the value of X using an output function β to obtain an output value.

Definition 6.1.1.
An *X-Machine* is a 10-tuple $M = (X, Y, Z, \alpha, \beta, Q, \Phi, \mathbf{F}, I, T)$, where
1. X is the *fundamental data set* that the machine operates on.
2. Y and Z are the *input* and the *output sets*, respectively.
3. α and β are the *input* and the *output (partial) functions* respectively, used to convert the input and the output sets into, and from, the fundamental set:
$\alpha: Y \to X, \quad \beta: X \to Z$
4. Q is the (finite) *set of states*.
5. Φ is the *type* of M , a set of relations on X:
$\Phi: P\ (X \to X)$
The type of the machine is the class of partial functions that constitute the elementary operations that the machine is capable of performing. P A denotes the power set of A and $(X \to X)$ denotes the set of all, possibly partial, functions from X to X. Φ is viewed as an abstract alphabet. Φ may be infinite, but only a finite subset Φ' of Φ is used (this is because M has only a finite number of edges despite the infinite number of labels available).
6. **F** is the *next-state partial function*:
$\mathbf{F}: Q \times \Phi \to P\ Q$
F is often described by means of a state-transition diagram.

7. *I* and *T* are the sets of *initial* and *terminal* (or *final*) *states* respectively:
$I \subseteq Q$, $T \subseteq Q$

Before we continue, we make some simple observations. It is sometimes helpful to think of an X-machine as a finite state machine with the arcs labelled by functions from the type Φ. As we shall define formally later, a computation takes the form of a traversal of a path in the state space and the application, in turn, of the path labels (which represent basic processing functions or relations) to an initial value of the data set *X*. Thus the machine transforms values of its data set according to the functions called during the state space traversal. The role of the input and output encoding relations is not crucial for many situations but it does provide a general interface mechanism that is useful in a number of applications.

Given an X-machine *M* as above, we can convert it into a finite state machine $A = (\Phi, Q, F, I, T)$ by treating the elements of Φ as abstract input symbols. We are, in effect, "forgetting" the *X* set and its structure and the semantics of the elements of Φ. We call this the *associated automaton* of *M* .

In a similar way to the case for a finite state machine, we can define an arc or a path of an X-machine. As one might expect, a path is a connected sequence of arcs through the machine starting from one state and ending at another (possibly the same one).

Definition 6.1.2.
If $q, q' \in Q, \phi \in \Phi$ and $q' \in F(q, \phi)$, we say that ϕ is the *arc* from *q* to *q'*, represented thus:

$\phi: q \to q'$ or $q \xrightarrow{\phi} q'$

Definition 6.1.3.
If $q, q' \in Q$ are such that there exist $q_1, ..., q_{n-1} \in Q$ and $\phi_1, ..., \phi_n \in \Phi$ with
$\phi_1: q \to q_1, \phi_2: q_1 \to q_2, ..., \phi_n: q_{n-1} \to q'$
we say that we have a *path* $p = \phi_1 ... \phi_n$ from *q* to *q'* and write $p : q \to q'$.

A *successful path* is one that starts in an initial state (in *I*) and ends in a final one (from *T*).

Unlike finite state machines though, each path *p* will give rise to a (partial) function
$|p| = \phi_n \circ \phi_{n-1} \circ ... \circ \phi_1 : X \to X$
So we apply the functions labelling the path one after the other in the order specified by the path. It may be that the composition is not defined at some point, because the possible values of *X* computed at that stage do not lie within the domain of the next function to be applied. Under these conditions the path will not be successful after that state.

The function $|\delta|$, corresponding to the empty path δ, is the identity function on *X*.

The union of all the functions computed over successful paths will make up the behaviour of the machine. The behaviour is a process that converts values of *X* into new values from X but only using the successful paths and the compositions of the functions defined by these paths.

Definition 6.1.4.
M determines a machine relation $|M|: X \leftrightarrow X$ in the following way:

$x\,|M|\,x' \Leftrightarrow \exists\, q_i \in I,\ q_t \in T$ and $p: q_i \rightarrow q_t$ such that $|p|(x) = x'$.

$|M|$ is called the *behaviour* of M.

So x' is related to x with respect to the machine M if we can find a path from an initial state to a terminal state such that the function defined by this path translates x into x'.

To relate this idea to the overall operation of the machine as it relates to its environment, that is its input and output functions, we proceed as follows.
Given y $\in$ Y, the operations of the X-machine M on Y consist of:
1. Picking a path p, from a start state q_i ($q_i \in I$), to a final state q_t ($q_t \in T$), p: $q_i \rightarrow q_t$
2. Applying α to the input to convert it to the internal type X.
3. Applying $|p|$, if it is defined for α(y). Otherwise, go back to step 1.
4. Applying β to get the output.

Therefore, the operation can be summarized as $\beta(|p|(\alpha(y)))$. Note that in general the output may be non-deterministic in the sense that a set of outputs is produced from a given input.

Definition 6.1.5.
The composite relation f given by:

$f = \beta \circ |M| \circ \alpha : Y \leftrightarrow Z$, i.e.

$$\begin{tikzcd} Y \arrow[r, "\alpha"] & X \arrow[r, leftrightarrow, "|M|"] & X \arrow[r, "\beta"] & Z \end{tikzcd}$$

is called the *relation computed* by M .

The X-machine model is sufficiently general to model most common types of machine from finite state machines (where the memory is trivial) to Turing machines (where the memory is a model of the tape), see [28].

When considering sequences of inputs and sequences of outputs we will use seq(A) to denote the set of finite sequences with members in A. < > will denote the empty sequence and $\text{seq}_+(A) = \text{seq}(A) - \{< >\}$. The notation $a^* \in \text{seq}(A)$ will denote a particular example of such a sequence, a* :: b* the concatenation of sequences a* and b* and $(a^*)^n$ will denote the concatenation of a* with itself for n times, $(a^*)^0 = < >$.
For two sets $U, V \subseteq \text{seq}(A)$, $U \bullet V = \{a^* :: b^* \mid a^* \in U, b^* \in V\}$.
Also $U^n = \{a_1^* ::..:: a_n^* \mid a_1^*,..., a_n^* \in U\}$, $U^0 = \{< >\}$.

Example 6.1.1 (Eilenberg [28])
To demonstrate that the X-machine is a general model of computation we will show how Turing machines, pushdown automata and finite state machines are all special cases. First, we introduce some further notations. These allow us to define a number of very simple functions that add and remove, where possible, symbols from either end of a string.

Let A an alphabet and let $s \in \text{seq}(A)$, so s = a*. Then we define (see [28]) the functions

L_s, $\mathrm{R}_s : \text{seq}(A) \to \text{seq}(A)$ by

$\mathrm{L}_s(x) = s :: x$, $\mathrm{R}_s(x) = x :: s \ \forall\, x, s \in \text{seq}(A)$

and the partial functions L_{-s}, $\mathrm{R}_{-s} : \text{seq}(A) \to \text{seq}(A)$ by

$\mathrm{L}_{-s}(x) = s^{-1} x$ (i.e. domain $\mathrm{L}_{-s} = \{s\} \bullet \text{seq}(A)$ and $\mathrm{L}_{-s}(s :: x) = x \ \forall\, x \in$ domain L_{-s});

$\mathrm{R}_{-s}(x) = x :: s^{-1}$ (i.e. domain $\mathrm{R}_{-s} = \text{seq}(A) \bullet \{s\}$ and $\mathrm{R}_{-s}(x :: s) = x \ \forall\, x \in$ domain R_{-s});

Obviously, L_s composed with L_t, that is $\mathrm{L}_s \mathrm{L}_t$, equals $\mathrm{L}_{t :: s}$, $\mathrm{R}_s \mathrm{R}_t = \mathrm{R}_{s :: t}$,
$\mathrm{L}_{-s} \mathrm{L}_{-t} = \mathrm{L}_{-t :: s}$, $\mathrm{R}_{-s} \mathrm{R}_{-t} = \mathrm{R}_{-s :: t} \ \ \forall\, s, t \in \text{seq}(A)$.

We shall denote the identity function on seq(A) by $1 : \text{seq}(A) \to \text{seq}(A)$.

(i) *Turing machine*. Let A be any finite alphabet. Let $X = \text{seq}(A) \times \text{seq}(A)$. We define the set of basic functions Φ to be a set of functions of the following forms:

$\mathrm{L}_s \times \mathrm{L}_t$; $\mathrm{L}_{-s} \times \mathrm{L}_t$; $\mathrm{L}_s \times \mathrm{L}_{-t}$; $\mathrm{L}_{-s} \times \mathrm{L}_{-t}$

where s, t $\in$ seq(A). These functions correspond to the usual operations applied to the tape represented by X. The input code is

$\alpha : \text{seq}(A) \to X$ is defined as $\alpha(s) = (1, s)$, $s \in \text{seq}(A)$.

(ii) *Pushdown automaton*, [90]. Let A and B be any finite alphabets, the latter representing the values that can be stored on the pushdown stack. Let $X = \text{seq}(B) \times \text{seq}(A)$. The basic functions Φ will be a set of functions of the following forms:

$1 \times \mathrm{L}_{-s}$, $\mathrm{R}_t \times 1$, $\mathrm{R}_{-t} \times 1$; $s \in \text{seq}(A)$, $t \in \text{seq}(B)$.

Also, $\alpha(s) = (1, s)$, s $\in$ seq(A).

(iii) *Finite state machine*. Let *Input* and *Output* be two finite alphabets. The set $X = \text{seq}(Input) \times \text{seq}(Output)$. The basic functions Φ will be a set of functions of the following forms:

$\mathrm{L}_t \times \mathrm{L}_{-s}$

where $s \in \text{seq}(Input)$, $t \in \text{seq}(Output)$.
Also, $\alpha(s) = (1, s)$, $s \in \text{seq}(Input)$.

Deterministic X-machines

A *deterministic* X-machine is one in which there is *at most one* possible applicable transition for any state q and any $x \in X$.

Definition 6.1.6.
An X-machine M = (X, Y, Z, α, β, Q, Φ, **F**, I, T) is called *deterministic* if the following are true.
1. **F** maps each pair $(q, \phi) \in Q \times \Phi$ onto at most a single next state, i.e.
   F: $Q \times \Phi \to Q$
A partial function is used because every $\phi \in \Phi$ will not necessarily be defined as the label to an edge in every state.
2. I contains only one element, i.e. $I = \{q_0\}$.
3. If ϕ and ϕ' are distinct arcs emerging from the same state

$$\begin{tikzcd} & q_1 \\ q \arrow[ur, "\phi"] \arrow[dr, "\phi'"'] & \\ & q_2 \end{tikzcd}$$

then domain $\phi \cap$ domain $\phi' = \varnothing$

Note that requirements 1 and 2 above are equivalent to saying that the associated automaton of the machine is deterministic.

In general, these conditions do not always ensure that $f = \beta \circ |M| \circ \alpha$ is a partial function. However, this will be the case for the particular type of X-machines that this book focuses on.

6.2. Stream X-machines.

A number of important classes of X-machines have been identified and studied, see [28], [36]. Typically the classes are defined by restrictions on the underlying data set X and the type, Φ, of the machines. In particular, the stream X-machines class has been found to be extremely useful in practice and most of the theory developed so far has concentrated on this class. This is the only class of X-machines that we consider here.

As suggested by their name, stream X-machines are those in which the input and the output sets are streams of symbols. The input stream is processed in a straightforward manner, producing, in turn, a stream of outputs and a regularly updated internal memory state. The power of this subclass of X-machines is considerable and they are able to model many practical computing situations. There are some natural restrictions that must be placed on the form of the processing functions ϕ to ensure that the machines behave in a sensible way. A formal definition is given next.

Definition 6.2.1.
An X-machine M is called *stream X-machine* if:
1. The type of the machine is of the form:
   $X = \text{seq}(\textit{Output}) \times \textit{Memory} \times \text{seq}(\textit{Input})$
where *Input*, *Memory*, *Output* are sets of data; *Input* and *Output* are finite, *Memory*

may be infinite. *Input* and *Output* are called the input and the output *alphabets* of the machine respectively, *Memory* is the machine's memory.

The processing functions that will form the set Φ will operate by reading the *current input* and the *current memory* value and then calculate a *new memory* value and an *output* value. The machine moves to a new state at the same time.

2. The overall inputs and outputs of the machine will be sets of input and output symbols respectively, i.e.

$Y = \mathrm{seq}(Input)$, $Z = \mathrm{seq}(Output)$

3. The input code α: $\mathrm{seq}(Input) \rightarrow \mathrm{seq}(Output) \times Memory \times \mathrm{seq}(Input)$ will define the *initial memory value* $m_0 \in Memory$, i.e. $\forall\, s^* \in \mathrm{seq}(Input)$

$\alpha(s^*) = (<>, m_0, s^*)$

Thus α takes a sequence of inputs and transforms them into a value of X by forming a triple with the empty output word and the initial memory value.

The output code β: $\mathrm{seq}(Output) \times Memory \times \mathrm{seq}(Input) \rightarrow \mathrm{seq}(Output)$ will extract the output sequence of any data value in X with empty input sequence, i.e. $\forall\, g^* \in \mathrm{seq}(Output)$,
$m \in Memory$, $s^* \in \mathrm{seq}(Input)$

$$\beta(g^*, m, s^*) = \begin{cases} g^*, & \text{if } s^* = <> \\ \perp, & \text{otherwise} \end{cases}$$

The symbol $\perp$ indicates that the function is undefined in these cases.

The output function just "prints" out the output string provided the computation has stopped with no further input waiting to be processed.

4. Each processing function $\phi \in \Phi$,

ϕ: $\mathrm{seq}(Output) \times Memory \times \mathrm{seq}(Input) \rightarrow \mathrm{seq}(Output) \times Memory \times \mathrm{seq}(Input)$

will process (and discard) an input symbol, produce an output symbol and may change the memory, i.e.

$\forall\, g^* \in \mathrm{seq}(Output), m \in Memory \cdot \phi(g^*, m, < >) = \perp$

and

$\forall\, m \in Memory, s \in Input \cdot$ either

$\forall\, g^* \in \mathrm{seq}(Output), s^* \in \mathrm{seq}(Input) \cdot \phi(g^*, m, s :: s^*) = \perp$ or

$\exists\, m' \in Memory, g \in Output$ (that depend on m and s) $\cdot$

$\forall\, g^* \in \mathrm{seq}(Output), s \in \mathrm{seq}(Input) \cdot \phi(g^*, m, s :: s^*) = (g^* :: g, m', s^*)$

Thus each basic function ϕ is such that it reads the current memory value, reads the current input value and generates a new memory value and an output value at each operation. As a stream of inputs arrive the basic functions will operate one by one if, at each stage, the pair of memory and input values are in the domain of a basic function at that state. Otherwise the machine halts.

A stream X-machine will be denoted by a tuple

$M = (Input, Output, Q, Memory, \Phi, \mathbf{F}, q_0, m_0, T)$.

Stream X-machines were first introduced by Laycock [35]. As pointed out in the first chapter, any processing function ϕ of a stream X-machine is completely determined by the values of g and m' above (if any), so, for the sake of simplicity, it will be referred to as:

$\phi: Memory \times Input \rightarrow Output \times Memory$

and defined in the form:

$\phi(m, s) = (g, m')$

So, starting the machine in a given initial state and an initial memory value and supplying it with a stream of inputs it will generate a corresponding stream of outputs.

Using this notation, the (partial) function produced by a path $p = \phi_1 ... \phi_n$ of a stream X-machine can be written as:

$|p|: Memory \times \text{seq}(Input) \rightarrow \text{seq}(Output) \times Memory$, where

$|p|(m, s^*) = (g^*, m') \Leftrightarrow \exists\, s_1,..., s_n \in Input, g_1,..., g_n \in Output, m_2,..., m_n \in Memory$ with

$s^* = s_1::...:: s_n$, $g^* = g_1::...:: g_n$ and $\forall$ i = 1...n, $\phi_i(m_i, s_i) = (g_i, m_{i+1})$, where $m_1 = m$ and $m_{n+1} = m'$. The partial function corresponding to the empty path δ will be defined by $|\delta|(m, < >) = (< >, m)$, where $m \in Memory$.

The string of inputs will force the machine to traverse a path in a deterministic machine or halt at the first state where no basic function is defined for the current input and current state. As it traverses the path it will generate a string of (internal) memory values and a string of outputs.

In our description we will use the projection functions $\pi_1, \pi_2, ..., \pi_n$ where

$\pi_1: A_1 \times A_2 \times ... \times A_n \rightarrow A_1$,

$\pi_2: A_1 \times A_2 \times ... \times A_n \rightarrow A_n$,

$\pi_n: A_1 \times A_2 \times ... \times A_n \rightarrow A_n$,

and $A_1, A_2, ..., A_n$ are sets.

Thus if $|p|(m, s^*) = (g^*, m')$ then the output g^* and the new memory value m' can be referred to as $\pi_1(|p|(m, s^*))$ and $\pi_2(|p|(m, s^*))$ respectively.

Also, given an initial memory value m_0, the relation f: seq($Input$) $\leftrightarrow$ seq($Output$) computed by the machine can be defined by:

$s^* f g^* \Leftrightarrow \exists\, q_i \in I, q_t \in T$ and $p: q_i \rightarrow q_t \cdot \pi_1(|p|(m_0, s^*)) = g^*$

All transitions of the machine M can be fully described by a relation

$[M]_0: Q \times Memory \times Input \leftrightarrow Output \times Q \times Memory$ defined by

$(q, m, s)\ [M\quad]_0\ (g, q', m') \Leftrightarrow \exists$ an arc ϕ: $q \rightarrow q' \cdot |\phi|(m, s) = (g, m')$.

Thus, the entire computation of the machine M can be described by a relation
$[M\quad]: Q \times Memory \times \mathrm{seq}(Input) \leftrightarrow \mathrm{seq}(Output) \times Q \times Memory$ defined by:
$\forall\, q \in Q, m \in Memory \cdot (q, m, < >)\ [M\quad]\ (< >, q, m)$
$\forall\, q, q' \in Q, m, m' \in Memory, s \in Input, s^* \in \mathrm{seq}(Input), g \in Output, g^* \in \mathrm{seq}(Output)$
$(q, m, s :: s^*)\ [M\quad]_0\ (g :: g^*, q', m') \Leftrightarrow \exists\, q'' \in Q, m'' \in Memory \cdot$
$(q, m, s)\ [M\quad]_0\ (g, q'', m'')$ and $(q'', m'', s^*)\ [M\quad]\ (g^*, q', m')$

In other words, $(q, m, s^*)\ [M\quad]\ (g^*, q', m')$ if and only if $\exists$ a path p: $q \rightarrow q'$ with $|p|(m, s^*) = (g^*, m')$.

$[M\quad]_0$ will be called the *transition relation* of M and $[M\quad]$ the *extended transition relation* of M .

Thus, as far as the machine computation is concerned, a stream X-machine behaves identically to an infinite state machine with outputs whose state set is $Q \times Memory$. However, there is one important difference which lies in their architecture: an X-machine is made up of a *finite* number of elements (the basic processing functions) which are combined by means of a *finite* state machine-like diagram. This feature will be of crucial importance when it comes to testing.

As defined previously, a deterministic stream X-machine will be one in which there is a single initial state (i.e. $I = \{q_0\}$) and there is at most one possible transition for any triplet $q \in Q, m \in Memory$, $s \in Input$, thus $[M\quad]_0$, $[M\quad]$ and f will be (partial) functions rather then relations. This will almost always be the case in practical applications and will be the assumption we will be making throughout our theoretical discussion.

Two concepts that will be needed in our description will be those of *reachable state* and *attainable memory.* A state q is called reachable if the machine can be brought into the state q from the initial state and the initial memory value. An unreachable state can never occur if we always start the machine up in the intital state with the initial memory value. All memory values that result from such a computation will be called attainable in q.

Definition 6.2.2.
A state q is called *reachable* if $\exists\, s^* \in \mathrm{seq}(Input), m \in Memory, g^* \in \mathrm{seq}(Output)$ and p: $q_0 \rightarrow q$ a path from the initial state q_0 to q with $|p|(m_0, s^*) = (g^*, m)$.

A machine in which all states are reachable will, itself, be called *reachable*. Clearly, if a state is not reachable it can be removed along with all arcs that emerge from or arrive to it without affecting the function computed by the machine.

Definition 6.2.3.
Let $q \in Q$. Then $m \in$ *Memory* is called *attainable in q* if and only if $\exists\, s^* \in \mathrm{seq}(Input)$ and a path p: $q_0 \rightarrow q$ with $\pi_2(|p|(m_0, s^*)) = m$.

Definition 6.2.4.
For any $q \in Q$ we define a subset of memory related to q by
$Mattain_q(M \quad) = \{m \in Memory : \exists\, s^* \in \mathrm{seq}(Input)$ and a path p: $q_0 \rightarrow q$ such that $\pi_2(|p|(m_0, s^*)) = m\}$.

Thus $Mattain_q(M \quad)$ is the set of memory values that can be taken by the machine whenever it is in state q.

Obviously, if q is reachable then $Mattain_q(M \quad)$ is not empty. The union of all sets $Mattain_q(M \quad)$ will make up the attainable memory of the machine.

Definition 6.2.5.
The subset of memory defined by

$$Mattain(M \quad) = \bigcup_{q \in Q} Mattain_q(M \quad)$$

is called the *attainable memory* of M .

Stream functions.

The next section will provide a characterization of the (partial) function computed by stream X-machines.

Definition 6.2.6.
Let f: $\mathrm{seq}(Input) \rightarrow \mathrm{seq}(Output)$ be a (partial) function. Then f is called a *weak (partial) stream function* if the following are true.
1. $\forall\, s^* \in$ domain f such that length($f(s^*)$) = length(s^*), where length(s^*) denotes the number of elements of the sequence s^*.
2. $\forall\, s^*, t^* \in \mathrm{seq}(Input)$, if $t^*, t^* :: s^* \in$ domain f then $\exists\, g^* \in \mathrm{seq}(Output)$ such that $f(t^* :: s^*) = f(t^*) :: g^*$.

Thus stream functions preserve the length of a sequence in the sense that the length of the resulting sequence equals the length of the sequence supplied to the function and it “respects” the concatenation operator in a natural way.

Theorem 6.2.1.
Let f: $\mathrm{seq}(Input) \rightarrow \mathrm{seq}(Output)$ be a (partial) function. Then f is a weak (partial) stream function if and only if there exists a stream X-machine that computes f.

Proof:
(i) Suppose that we have a streamX-machine which computes f, then length($f(s^*)$) = length(s^*) is obvious. Also, the second requirement is a result of the machine being deterministic.
(ii) Suppose that f is a weak (partial) stream function. A two state stream X-machine that computes f can be easily constructed. This will have the state transition diagram as in Figure 6.1 and *Memory* = seq(*Input*). ϕ_1 will process all pairs $(t^*, s) \in$ *Memory* $\times$ *Input* for which $t^* :: s \in$ domain f and ϕ_2 those pairs for which $t^* :: s \notin$ domain f. *A* will be the only terminal state of the machine. *A* will also be the initial state of the machine if and only if $< > \in$ domain f.

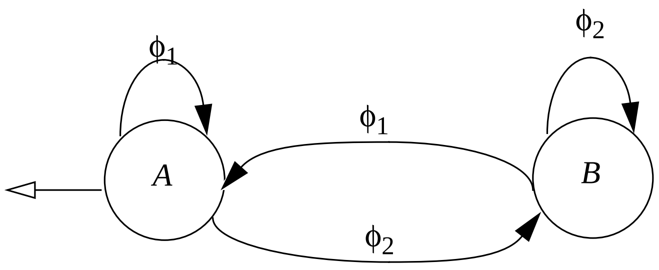

Figure 6.1 A two state stream X-machine computing a weak partial stream function.

Of special interest to us will be machines whose states are all terminal (i.e. $T = Q$). If this is the case then f will be defined by:

$$f(s^*) = g^* \Leftrightarrow \exists\, q \in Q \text{ and } p: q_0 \rightarrow q \cdot \pi_1(|p|(m_0, s^*)) = g^*$$

or alternatively by:

$$f(s^*) = (\pi_1 \circ [M\ \ ])(q_0, m_0, s^*)$$

Definition 6.2.7.
Let f: seq(*Input*) $\rightarrow$ seq(*Output*) be a weak (partial) stream function. Then f is called a *(partial) stream function* if

$$\forall\, s^*, t^* \in \text{seq}(Input) \cdot \text{if } t^* :: s^* \in \text{domain } f \text{ then } t^* \in \text{domain } f$$

Theorem 6.2.2.
Let f: seq(*Input*) $\rightarrow$ seq(*Output*) be a (partial) function. Then f is a (partial) stream function if and only if there exists a stream X-machine with all states terminal that computes f.

Proof:
(i) If f is computed by such a stream X-machine then it is obviously a stream function.
(ii) Suppose that f is a stream function. It is easy to construct a single state stream X-machine as in Figure 6.2 that computes f. If *Memory* = seq(*Input*) then ϕ_1 will process all pairs $(t^*, s) \in$ *Memory* $\times$ *Input* for which $t^* :: s \in$ domain f.

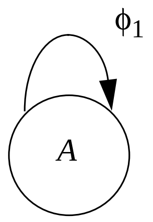

Figure 6.2 A single state stream X-machine.

Example 6.2.1.
Suppose that *M* is a 3 state machine as shown in Figure 6.3 (the initial state is marked with an in-arrow).

Input = *Output* = {0, 1}, *Memory* = {10, 11} and $m_0 = 10$

M has four processing functions, ϕ_1, ϕ_2, ϕ_3, ϕ_4 where

$\phi_1(10, 0) = (1, 11)$
$\phi_2(11, 0) = (1, 10)$
$\phi_3(11, 0) = (1, 11)$
$\phi_4(11, 1) = (0, 11)$

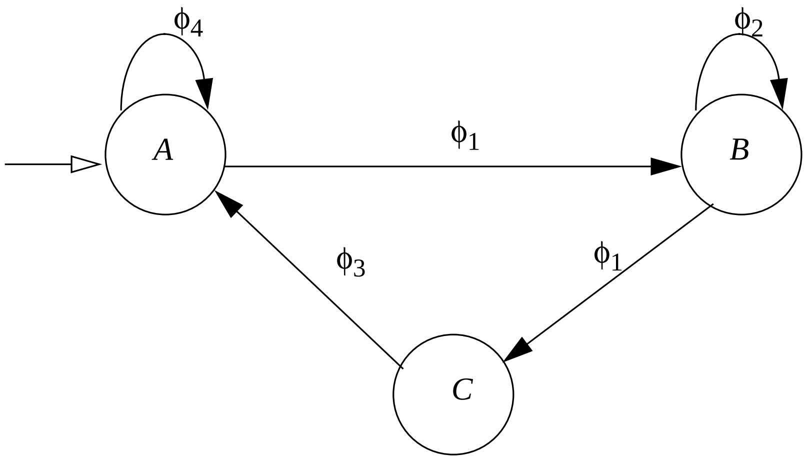
$$\begin{tikzcd}
\arrow[r] & A \arrow[loop above, "\phi_4"] \arrow[rr, "\phi_1"] & & B \arrow[loop above, "\phi_2"] \arrow[dl, "\phi_1"] \\
& & C \arrow[ul, "\phi_3"] &
\end{tikzcd}$$

Figure 6.3.

It is easy to see that if *A* is the only terminal state of the machine then *M* will compute the following partial function *f*:

$f(< >) = < >$
$f(0 :: 0 :: 0 :: 0 :: 1^n) = 1:: 1:: 1:: 1:: 0^n, n \geq 0$

On the other hand, if all states are terminal then *f* will be of the form:

$f(< >) = < >$
$f(0) = 1$
$f(0 :: 0) = 1:: 1$

$f(0 :: 0 :: 0) = 1 :: 1 :: 1$

$f(0 :: 0 :: 0 :: 0 :: 1^n) = 1 :: 1 :: 1 :: 1 :: 0^n, n \geq 0$

In the first case f is a *weak partial stream function*, in the second case f is a *partial stream function*.

Aside. When we consider the issue of testing in Chapter 7 we shall assume that all the states of the machine are terminal - thus the set T will be ignored unless it has a particular significance for our theoretical discussion. This means that all paths that emerge from the initial state will be successful so their corresponding outputs are observable to the user. This is particularly useful when it comes to testing since all outputs - even those that correspond to intermediate computations - are observable thus providing more information about the system behaviour. Obviously the extra outputs can be filtered out once the system has passed all tests.

Before we go any further, we give one more definition.

Definition 6.2.8.
Let f: seq($Input$) $\rightarrow$ seq($Output$) be a (partial) function and let $t^* \in$ domain f. Then we can define a (partial) function f: seq($Input$) $\rightarrow$ seq($Output$) by

$f_{t^*}(s^*) = f(t^*)^{-1} :: f(t^* :: s^*)$, where $t^* :: s^* \in$ domain f

For $a^*, b^* \in$ seq(A) if there exists c^* with $a^* :: c^* = b^*$ then $a^{*-1} :: b^* = c^*$. It is easy to see that f_{t^*} is a (partial) stream function. In terms of stream X-machines f_{t^*} admits the following straightforward interpretation. If $t^* \in$ domain f then there exist $q \in Q$, $m \in Memory$ and a path $p: q_0 \rightarrow q$ with $\pi_2(|p|(m_0, s^*)) = m$. Then f_{t^*} is the (partial) function computed by M if the initial state and memory are q and m respectively.

Example 6.2.2.
For the stream function f of Example 6.2.1 and $t^* = 0 :: 0 :: 0$ then f_{t^*} is defined by:

$f_{t^*}(< >) = < >$

$f_{t^*}(0 :: 1^n) = 1 :: 0^n, n \geq 0$

6.3. Stream X-machine refinement.

If stream X-machines are to be useful as a tool for specification, there needs to be some way of developing existing machines into more complex and more detailed versions without starting anew with each modification. The remaining part of this chapter will investigate the concept of refinement in the context of X-machines. Several types of refinement have been studied [57], [91], [92], we will only present one of them, one that is potentially the most useful in practical applications.

When we say that a machine M is a refinement of another machine M' we mean that the new machine is more complex than the original but it preserves in some sense its functional properties. In other words, the refined machine will do everything that the original did plus some extra functionality. The internal structure of M' may differ from that of M in various ways. Obviously, we will be interested in methods of constructing M' starting from the architecture of M, so that the internal structure of M is preserved in some sense and the construction of the new model will not start anew with each modification.

The following two sections will define the concept of refinement in terms of the input-output behaviour of two machines. In general terms, given two stream X-machines M and M', we say that M' is a *functional refinement* of M if:

each input (or output) sequence of M can be mapped onto at least one input (or output (sequence) of M';
each input/output transformation of M can be mapped onto at least one input/output transformation of M'.

In order to give a more rigorous interpretation of these 2 requirements we introduce some preliminary concepts.

6.3.1. Alphabet covering.

Definition 6.3.1.1.
Let A be a set and P a subset of seq(A). Then P is called a *prefix* if the following is true.
$\forall\ a^*, b^* \in$ seq(A) such that if $a^*, a^* :: b^* \in P$ then $b^* = < >$.

Obviously, if the empty sequence $< > \in P$ then $P = \{ < > \}$. We will say that P is a *non-trivial prefix* if P is non-empty and $P \neq \{< >\}$.

Definition 6.3.1.2.
Let A and A' be two alphabets and let u: seq(A') $\rightarrow A$ be a (partial) surjective function. If the domain of u is a non-trivial prefix then we say that seq(A') *covers* seq(A) w.r.*t. u*. Also, u will be called an *alphabet covering* and the domain of u the *domain of the covering*.

Lemma 6.3.1.1.
If u is an alphabet covering, as above, then the following is true.
$\forall\ a_1^*, a_2^*, ..., a_n^*, b_1^*, b_2^*, ..., b_m^* \in$ domain u
if $a_1^* :: a_2^* :: ... :: a_n^* = b_1^* :: b_2^* :: ... :: b_m^*$ then $n = m$ and $a_1^* = b_1^*, a_2^* = b_2^*, ..., a_n^* = b_n^*$.

Proof:
This follows by induction on n since domain u is a prefix.

Then we can define a partial surjective function u^*: seq(A') $\rightarrow$ seq(A) by:

$$u^*(a^*) = \begin{cases} < >, \text{ if } a^* = < > \\ u(a_1^*) ::..:: u(a_n^*), \text{ if } \exists\, n \geq 1, a_1^*, ..., a_n^* \in \text{ domain } u \text{ with } a^* = a_1^*::..:: a_n^* \\ \perp, \text{ otherwise} \end{cases}$$

Clearly, domain u^* = seq(domain u).

We illustrate the above concept with two examples.

Example 6.3.1.1.
Let $A \subseteq A'$ be two finite alphabets and let u: seq(A') $\rightarrow A$ be a surjective partial function with domain $u \subseteq$ seq(A' - A) $\bullet$ A that extracts the element of A from any sequence for which u is defined, i.e.
$\forall\, a \in A, b^{*\prime} \in$ seq(A' - A) $\cdot$ if $b^{*\prime}\, a \in$ domain u then $u(b^{*\prime}\, a) = a$
We will call u a *filter* of A w.r.t. A'.
It is easy to see that the domain of u is a prefix, seq(A') covers seq(A) w.r.*t.* u and that domain u^* $\subseteq$ (seq(A') $A \cup \{< >\}$ *)* and u^* will extract all elements of A from any sequence in the domain of u^*.

For example let $A = \{0, 1\}$ and $A' = \{0, 1, a, b\}$. Then v: seq(A') $\rightarrow A$ defined by
$v(s^*:: 0) = 0, s^* \in$ seq($\{a, b\}$)
$v(s^*::1) = 1, s^* \in$ seq($\{a, b\}$)
is an alphabet covering.

Also, for any k $\geq$ 0, u_k: seq(A') $\rightarrow A$ defined by
$u_k(a^n :: 0) = 0$, n $\geq$ k
$u_k(b^n :: 1) = 1$, n $\geq$ k
are alphabet coverings.

Example 6.3.1.2.
Let A and A' be two alphabets, $b \in A'$ and let w: seq(A' - $\{b\}$) $\rightarrow A$ be a (partial) surjective function. Then seq(A') covers seq(A) w.r.*t.* u, where u is defined by
$u(a^{*\prime}::b) = w(a^{*\prime}), a^{*\prime} \in$ seq(A' - $\{b\}$).

We illustrate this type of covering with the following practical application. Let CHARS be the set of all alphanumeric characters and WORDS_n the set of all words of at most n characters, i.e. WORDS_n = {‘a_1 a_2 ... a_j’ | $a_1, a_2, ..., a_j \in$ CHARS, $0 \leq j \leq n$}.
‘a_1 a_2 ... a_j’ denotes the word composed of characters a_1, a_2, ... a_j.
If j = 0 then ‘a_1 a_2 ... a_j’ = ‘’, the empty word.

We assume that each such word can be entered from the keyboard as a sequence of characters followed by *enter* and that the keys available are all character keys, *backspace* and *enter.* If more than n characters have been entered, then the rest will be ignored, unless one or more of the first n characters have been deleted. This process can be described by a partial function

u_n: seq(CHARS ∪ {*backspace*, *enter*}) → WORDS$_n$

with domain u_n = seq(CHARS ∪ {*backspace*}) • {*enter*} defined by

u_n(*enter*) = ‘’

$$u_n(a^* :: \textit{backspace} :: \textit{enter}) = \begin{cases} \text{‘}a_1 \ldots a_{j-1}\text{’ if } u(a^* :: \textit{enter}) = \text{‘}a_1 \ldots a_j\text{’ with } 0 < j \le n \\ \text{‘’, if } u(a^* :: \textit{enter}) = \text{‘’} \end{cases}$$

where a* ∈ seq(CHARS ∪ {*backspace*})

$$u_n(a^* :: b :: \textit{enter}) = \begin{cases} \text{‘}a_1 ::...:: a_j :: b\text{’ if } u(a^* :: \textit{enter}) = \text{‘}a_1 ::...:: a_j\text{’ with } 0 \le j < n \\ \text{‘}a_1 ::...:: a_j\text{’, if } u(a^* :: \textit{enter}) = \text{‘}a_1 ::...:: a_j\text{’ with } j = n \end{cases}$$

where a* ∈ seq(CHARS ∪ {*backspace*}), b ∈ CHARS

It is easy to see that seq(CHARS ∪ {*backspace*, *enter*}) covers seq(WORDS$_n$) w.r.t. u_n.

If there is no restriction on the number of characters that a word can have then the corresponding alphabet covering will result from the definition above for n = ∞.

6.3.2. Functional covering.

Our definition of refinement is aimed at meeting the two requirements stated above while being precise enough to be both meaningful and usable in examples. This is given next.

Definition 6.3.2.1.
Let *f*: seq(*Input*) → seq(*Output*) and *f* ': seq(*Input*') → seq(*Output*') be two (partial) stream functions such that seq(*Input*') covers seq(*Input*) w.r.t. *u* and seq(*Output*') covers seq(*Output*) w.r.t. *v*. Then we say that *f* ' is a *refinement of f* w.r.t. (*u*, *v*) if the following are true:

1. *f '* (domain *u**) ⊆ domain *v**
2. The following diagram commutes:

$$\begin{tikzcd} \text{seq}(\textit{Input}') \arrow[r, "f'"] \arrow[d, "u^*"'] & \text{seq}(\textit{Output}') \arrow[d, "v^*"] \\ \text{seq}(\textit{Input}) \arrow[r, "f"] & \text{seq}(\textit{Output}) \end{tikzcd}$$

Definition 6.3.2.2.
Let *M* and *M* ' two X-machines and *f* and *f* ' respectively the functions they compute. Then we say that *M* ' is a *functional refinement* of *M* w.r.t. (*u*, *v*) if *f* ' is a refinement of *f* w.r.t. (*u*, *v*).

Lemma 6.3.2.1.
M ' is a functional refinement of M w.r.t. (u, v) if and only if the following are true.
1. $\forall$ p' a path in M ' that starts in q_0' with $\pi_1(|p'|(m_0', s^{*\prime})) = g^{*\prime}$,

   $s^{*\prime} \in$ domain $u^* \Leftrightarrow g^{*\prime} \in$ domain v^*.
2. $\forall$ $s^* \in \mathrm{seq}_+(Input)$ the following requirements are equivalent.

   i) $\exists$ $g^* \in \mathrm{seq}_+(Output)$ and p a path in M that starts in q_0 with $\pi_1(|p|(m_0, s^*)) = g^*$

   ii) $\forall$ $s^{*\prime} \in u^{*-1}(s^*)$ $\exists$ $g^{*\prime} \in$ domain v^* and a path p' in M ' that starts in q_0' such that $\pi_1(|p'|(m_0', s^{*\prime})) = g^{*\prime}$

where g^* and $g^{*\prime}$ above satisfy $v^*(g^{*\prime}) = g^*$.

Proof:
This follows from the definition of functional covering and that of the function computed by a stream X-machine.

Thus everything that is processed by M can be processed by M ' with essentially the same result. For a discussion of related topics with respect to finite state machines see [25].

Example 6.3.2.1.
Let $Input = Output = \{0, 1\}$, $Input' = Output' = \{0, 1, a, b\}$ and let
f: seq($Input$) $\rightarrow$ seq($Output$) be the stream function from example 6.2.1, i.e.

$f(< >) = < >$
$f(0) = 1$
$f(0 :: 0) = 1 :: 1$
$f(0 :: 0 :: 0) = 1 :: 1 :: 1$
$f(0 :: 0 :: 0 :: 0 :: 1^n) = 1 :: 1 :: 1 :: 1 :: 0^n$, where $n \geq 0$

Then it is easy to see that for any $k \geq 0$, f_k' is a refinement of f w.r.t. (u_k, v), where u_k and v are those of Example 6.3.1.1 and f_k' is defined by:

$f_k'(a^{j1}) = b^{j1}$, $j_1 \geq 0$

$f_k'(a^{j1} :: 0 :: a^{j2}) = b^{j1} :: 1 :: b^{j2}$, $j_1 \geq k$, $j_2 \geq 0$

$f_k'(a^{j1} :: 0 :: a^{j2} :: 0 :: a^{j3}) = b^{j1} :: 1 :: b^{j2} :: 1 :: b^{j3}$, $j_1 \geq k$, $j_2 \geq k$, $j_3 \geq 0$

$f_k'(a^{j1} :: 0 :: a^{j2} :: 0 :: a^{j3} :: 0 :: a^{j4}) = b^{j1} :: 1 :: b^{j2} :: 1 :: b^{j3} :: 1 :: b^{j4}$, $j_1 \geq k$, $j_2 \geq k$, $j_3 \geq k$, $j_4 \geq 0$

$f_k'(a^{j1} :: 0 :: a^{j2} :: 0 :: a^{j3} :: 0 :: a^{j4} :: 0 :: b^{m1}) =$
$b^{j1} :: 1 :: b^{j2} :: 1 :: b^{j3} :: 1 :: b^{j4} :: 1 :: a^{m1}$, $j_1 \geq k$, $j_2 \geq k$, $j_3 \geq k$, $j_4 \geq k$, $m_1 \geq 0$

....

$f_k'(a^{j1} :: 0 :: a^{j2} :: 0 :: a^{j3} :: 0 :: a^{j4} :: 0 :: b^{m1} :: 1 :: \ldots :: b^{m_{n-1}} :: 1 :: b^{m_n}) =$
$b^{j1} :: 1 :: b^{j2} :: 1 :: b^{j3} :: 1 :: b^{j4} :: 1 :: a^{m1} :: 1 :: \ldots :: a^{m_{n-1}} :: 1 :: a^{m_n}$,
$j_1 \geq k$, $j_2 \geq k$, $j_3 \geq k$, $j_4 \geq k$, $m_1 \geq k$, ..., $m_{n-1} \geq k$, $m_n \geq 0$, $n \geq 0$

Before we go any further we state two results that we will use in the following sections.

Lemma 6.3.2.2.
If f' is a refinement of f w.r.t. (u, v) and $t^{*'} \in$ domain u^* then the following are true.
1. $f'(t^{*'}) \neq \perp$ if and only if $f(t^*) \neq \perp$, where $t^* = u^*(t^{*'})$.
2. If $f'(t^{*'}) \neq \perp$ then f_{t^*} is a refinement of $f'_{t^{*'}}$ w.r.t. (u, v), where $t^* = u^*(t^{*'})$.

Proof:
1. This follows from Definition 6.3.2.1.
2. If $s^{*'} \in$ domain u^* and $s^* = u^*(s^{*'})$ then $f(t^{*}:: s^*) = f(t^*) :: f_{t^*}(s^*)$ and $f'(t^{*'}:: s^{*'}) = f'(t^{*'}) :: f'_{t^{*'}}(s^{*'})$ hence $f(t^*) :: f_{t^*}(s^*) = v^*(f'(t^{*'})) :: v^*(f'_{t^{*'}}(s^{*'}))$

Lemma 6.3.2.3.
If f' is a refinement of f w.r.t. (u, v), $t^{*'} \in$ domain u^*, $s^{*'} \in$ domain u and M is a stream X-machine that computes f then the following are true.
1. $f'(t^{*'}) \neq \perp \Leftrightarrow \exists\, q_1 \in Q, m_1 \in Memory$ and a path p: $q_0 \rightarrow q_1$ in M with $\pi_2(|p|(m_0, t^*)) = m_1$, where $t^* = u^*(t^{*'})$.
2. If $f'(t^{*'}) \neq \perp$ then:
$f'_{t^{*'}}(s^{*'}) \neq \perp \Leftrightarrow \exists\, q_2 \in Q, m_2 \in Memory$ and an arc ϕ: $q_1 \rightarrow q_2$ with
$\phi(m_1, s) = (m_2, f_{t^*}(s))$,
where $t^* = u^*(t^{*'})$ and $s = u(s^{*'})$ and $q_1 \in Q, m_1 \in Memory$ are those from 1.

Proof:
Follows from lemma 6.3.2.2

6.3.3. Construction of refinement. State refinement.

Once the transformation has been defined in terms of the input-output behaviour of the two machines, we turn our attention to the construction of the refined machine M'. Intuitively M' will have a larger state space and memory that will enable it to perform the more complex functionality. On the other hand, since the functionality of the original machine M will be preserved one would expect that it should be possible to find a systematic way of developing the architecture of M into an architecture that will correspond to the refined input-output behaviour. The following result (Eilenberg [28]) provides us with a basic idea for our construction.

Lemma 6.3.3.1.
Let A be a set and $P \subseteq \mathrm{seq}(A)$. Then P is a non-trivial prefix if and only if P is the behaviour of a (possible infinite) state machine that has a single final state q_t and no arcs emerge from q_t.

So it appears that all we need to do is refining (in the sense of the commutative diagram of Definition 6.3.2.1) each arc of M to a stream X-machine with a single terminal state (i.e. this is computationally equivalent to an infinite finite state machine) and replacing all arcs with their corresponding refining machine.

However, this idea, although straightforward, has no immediate application, the principal reason being that the construction could result in a non-deterministic machine. Suppose, for example, that M is a machine with *Input* = *Output* = {0, 1} for which there are two arcs ϕ_1 and ϕ_2 emerging from the same state, where

$\phi_1(m, 0) = (0, m), m \in \mathit{Memory}$

$\phi_2(m, 1) = (1, m), m \in \mathit{Memory}$

that *Input*' = *Output*' = {0, 1, a} and that the alphabet refinements u and v are defined by:

$u(\mathrm{a} :: 0) = v(\mathrm{a} :: 0) = 0$

$u(\mathrm{a} :: 1) = v(\mathrm{a} :: 1) = 1$

Then ϕ_1 and ϕ_2 can be refined to M_1 and M_2 shown in figure 6.4 (a) and (b), with ϕ_3 defined by

$\phi_3(m, \mathrm{a}) = (\mathrm{a}, m), m \in \mathit{Memory}$

thus the construction suggested above will lead to a non-deterministic machine, see Figure 6.4 (c).

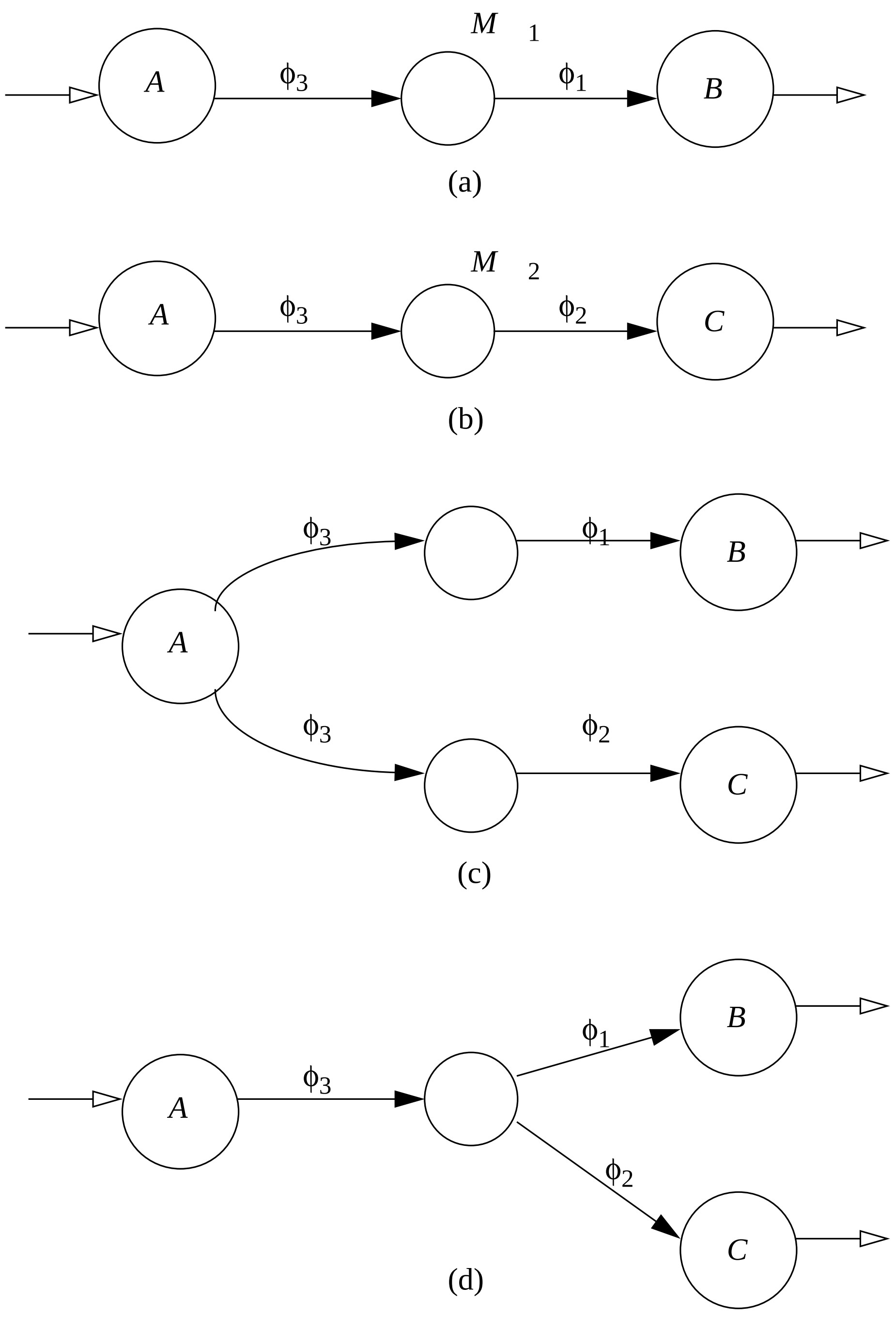

Figure 6.4 Non-determinism and refinement.

However, there is a way around this problem: rather than replacing each arc with a machine with a single terminal state, we replace all the arcs that emerge from a state q with a single machine that has a terminal state for each such arc, see Figure 6.4 (d). This idea will be pursued in what follows and will form the basis of the concept of *state refinement*.

So we are going to construct a *refining* machine $M'(q)$ for a given state q whose state set will be denoted by R_q , this will consist of a *copy* of q denoted by q' together with some new states needed to act as intermediate states along the new paths that will be replacing the original transitions from *q*. We also add a *copy* of all the states that these transitions point at. This second set of new states will be our set of terminal states, T_q . Then, each arc labelled by a ϕ in the original machine that leaves the state *q* will be replaced by a path in *M* '(*q*) that will correspond to the original transition *modulo* the covering relationships between the input and output alphabets. This is illustrated in Figure 6.5.

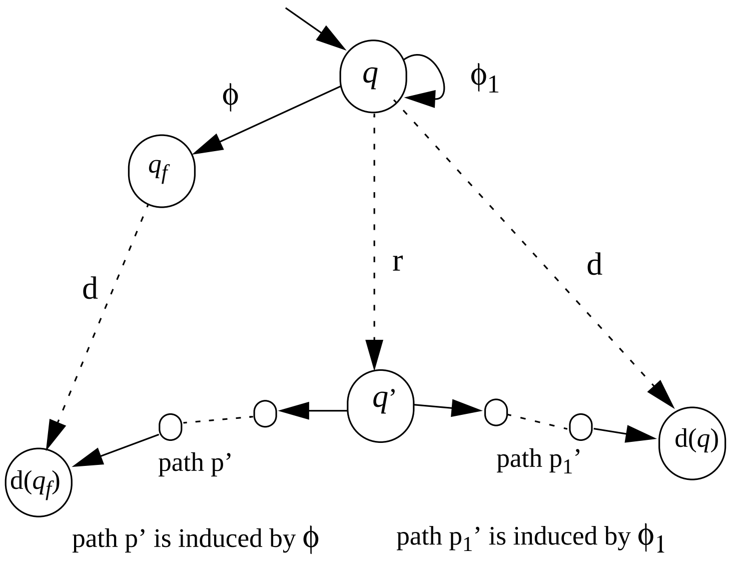

Fig. 6.5 The state refinement relationship.

Notation. In our description, for n (partial) functions $f_1: A_1 \rightarrow B_1, ..., f_n: A_n \rightarrow B_n$, $\{f_1, ..., f_n\}: A_1 \times ... \times A_n \rightarrow B_1 \times ... \times B_n$ will be a partial function defined by

$$\{f_1, ..., f_n\}(a_1, ..., a_n) = (f_1(a_1), ..., f_n(a_n)), a_1 \in A_1, ..., a_n \in A_n$$

First, we prepare the ground for our formal construction.
Let *M* = (*Input*, *Output*, *Q*, *Memory*, **Φ**, **F**, q_0, m_0) be a stream X-machine. Let *Input*' and *Output*' be two alphabets so that seq(*Input*') covers seq(*Input*) w.r.t. *u* and seq(*Output*') covers seq(*Output*) w.r.t. *v*. The essential strategy is to consider a state $q \in Q$ and to replace each arc leaving *q* with a submachine. So that, if ϕ_1 is a transition from *q* to *q'* then we introduce a set of states $q^1{}_1, q^1{}_2, ..., q^1{}_{n1}$ of the submachine with a path through them replacing ϕ_1. We do this for each transition ϕ_i that leaves *q*. The states of the refining machine for *q* will consist of all these states together with the state *q*. Of these states the states $q^i{}_{ni}$ for each ϕ_i will be designated terminal states in the refining machine for *q*. We need to construct a copy, *L* of *Q* to provide for the eventual pasting of the refining machine into the original. This will be done by recognising that the terminal states $q^i{}_{ni}$ will be *merged* with their corresponding states in *Q* as we considered in Chapter 4.

So let K (the *key* states of the refinement) and L be two disjoint sets, $K \cap L = \emptyset$, with $\mathrm{Card}(K) = \mathrm{Card}(L) = \mathrm{Card}(Q)$ and let r: $Q \to K$ and d: $Q \to L$ be two bijections. Let also *Memory'* be a set for which there exists h: *Memory'* $\to$ *Memory* a surjective (partial) function whose domain is H.

Definition 6.3.3.1.
Then for any $q \in Q$, a stream X-machine
$M'(q) = (Input', Output', R_q, Memory', \Phi_q', \mathbf{F}_q, q', m_q', T_q)$ is called a *refining machine in q* w.r.t. (u, v) if the following are true.
1. The initial state is $q' = \mathrm{r}(q)$. The set of terminal states is a subset of L defined by:
   $T_q = \{\mathrm{d}(q_f): \exists$ an arc $\phi: q \to q_f$ in $M\}$.
2. There is no arc emerging from any terminal state in $M'(q)$.
3. If $E_q = R_q - T_q$ then $E_q \cap K = \{\mathrm{r}(q)\}$ and $E_q \cap L = \emptyset$. Thus the internal states of $M'(q)$ - i.e. those that are neither initial nor terminal - are not among the elements of K and L.
4. If $q = q_0$ then the initial memory m_q' satisfies $\mathrm{h}(m_q') = m_0$.
5. For any $q_f' \in R_q$, $m_i' \in \mathrm{H}$, $m_f' \in Memory'$, $s^{*\prime} \in \mathrm{seq}(Input')$, $g^{*\prime} \in \mathrm{seq}(Output)$ and p': $\mathrm{r}(q) \to q_f'$ a path in $M'(q)$ with $|p'|(m_i', s^{*\prime}) = (g^{*\prime}, m_f')$ the following are true.
   i) if $s^{*\prime} \in$ domain u then $q_f' \in T_q$, $m_f' \in \mathrm{H}$, $g^{*\prime} \in$ domain v.
   ii) if $s^{*\prime} \in \mathrm{seq}_+(Input')$ - domain u^* then $q_f' \in (R_q - T_q)$ and
$g^{*\prime} \in \mathrm{seq}_+(Output')$ - domain v^*.
6. For any $s \in Input$, $m_i \in Mattain_q(M)$, $m_i' \in \mathrm{h}^{-1}(m_i)$ the following requirements are equivalent.
   i) $\exists\, q_f \in Q$, $m_f \in Memory$, $g \in Output$ and $\phi: q \to q_f$ an arc in M such that $\phi(m_i, s) = (g, m_f)$.
   ii) $\forall\, s^{*\prime} \in u^{-1}(s)\ \exists\, q_f' \in T_q$, $m_f' \in \mathrm{H}$, $g^{*\prime} \in$ domain v and a path p' in $M'(q)$ with $|p'|(m_i', s^{*\prime}) = (g^{*\prime}, m_f')$,
where q_f, q_f', m_f, m_f', g and $g^{*\prime}$ above satisfy $\mathrm{d}(q_f) = q_f'$, $\mathrm{h}(m_f') = m_f$ and $v(g^{*\prime}) = g$.

Basically, requirement 5 states that only sequences of inputs that are in the domain of u will reach a terminal state and the corresponding output sequence will be in the domain of v. Requirement 6 ensures that for any arc $\phi: q_i \to q_f$ everything that is processed by ϕ will be processed by a path p': $\mathrm{r}(q_i) \to \mathrm{d}(q_f)$ in $M'(q)$ from the initial state to the appropriate terminal state with essentially the same result.

Definition 6.3.3.2.
Let M be a stream X-machine and $\{M'(q): q \in Q\}$ a set of refining machines w.r.t. (u, v) indexed by Q for which $(E_{q1} - \{\mathrm{r}(q_1)\}) \cap (E_{q2} - \{\mathrm{r}(q_2)\}) = \emptyset$ for any 2 distinct states, q_1 and q_2 of Q. Then a stream X-machine
$M' = (Input', Output', Q', Memory', \Phi', \mathbf{F}', q_0', m_0')$ is called a *state refinement* of M w.r.t. (u, v) if the following are true:
1. The state space is $Q' = \bigcup_{q \in Q} E_q$, where $\forall\, q \in Q$, $E_q = R_q - L$.

2. The initial state $q_0' = \mathrm{r}(q_0)$ is the initial state of $M'(q_0)$.

3. The initial memory m_0' is the initial memory of $M'(q_0)$.

4. $\Phi' = \bigcup_{q \in Q} \Phi'_q$

5. The next state (partial) function $\mathbf{F}': Q' \times \Phi' \rightarrow Q'$ is defined by

$$\mathbf{F}'(q', \phi') = \begin{cases} \mathbf{F}_q(q', \phi'), \text{ if } \mathbf{F}_q(q', \phi') \in E_q \\ \mathrm{r}(\mathrm{d}^{-1}(\mathbf{F}_q(q', \phi'))), \text{ if } \mathbf{F}_q(q', \phi') \in T_q. \\ \perp, \text{ if } \mathbf{F}_q(q', \phi') = \perp \end{cases}$$

where q is chosen such that $q \in E_q$.
Note that $\mathbf{F}'$ is well defined since $\{E_q : q \in Q\}$ is a partition of Q' indexed by Q.

Also, $\{M'(q): q \in Q\}$ will be called the *refining set of* M'.

Basically, the state transition diagram of M' will result by merging each terminal state of a refining machine with the appropriate key state, i.e. for any state $q \in Q$ each terminal state $\mathrm{d}(q)$ of $M'(q)$ will be merged with $\mathrm{r}(q)$, the initial state of $M'(q)$. Thus each arc $\phi: q \rightarrow q_1$ in M will give rise to a set of paths $\{p': \mathrm{r}(q) \rightarrow \mathrm{r}(q_1)\}$ in the refined machine M'. We illustrate the construction with two examples.

Example 6.3.3.1.
Suppose that M is the three state machine from Example 6.2.1 and that *Input'*, *Output'* , u_k and v are those of Example 6.3.1.1. Then $M'(A)$, $M'(B)$, $M'(C)$ shown in Figure 6.6 will be refining machines in A, B and C respectively w.r.t. (u_1, v).

We take $K = Q$, $L = \{A_1, B_1, C_1\}$, *Memory' = Memory*,
r and h are the identity functions and d is defined by:
$\mathrm{d}(A) = A_1$, $\mathrm{d}(B) = B_1$, $\mathrm{d}(C) = C_1$

$R_A = \{ A, D, E, A_1, B_1\}$; $T_A = \{ A_1, B_1\}$; $E_A = \{ A, D, E\}$.
$R_B = \{ B, F, B_1, C_1\}$; $T_B = \{ B_1, C_1\}$; $E_B = \{ B, F\}$.
$R_C = \{ C, G, G_1\}$; $T_C = \{ C_1\}$; $E_C = \{ C, G\}$.
The transition functions ϕ_5 and ϕ_6 are defined by:
$\phi_5(\mathrm{a}, m) = (m, \mathrm{b})$, $m \in$ *Memory*
$\phi_6(\mathrm{b}, m) = (m, \mathrm{a})$, $m \in$ *Memory*
The refined machine M' may be found in Figure 6.7. It is easy to see that M' computes f_1' from Example 6.3.2.1.

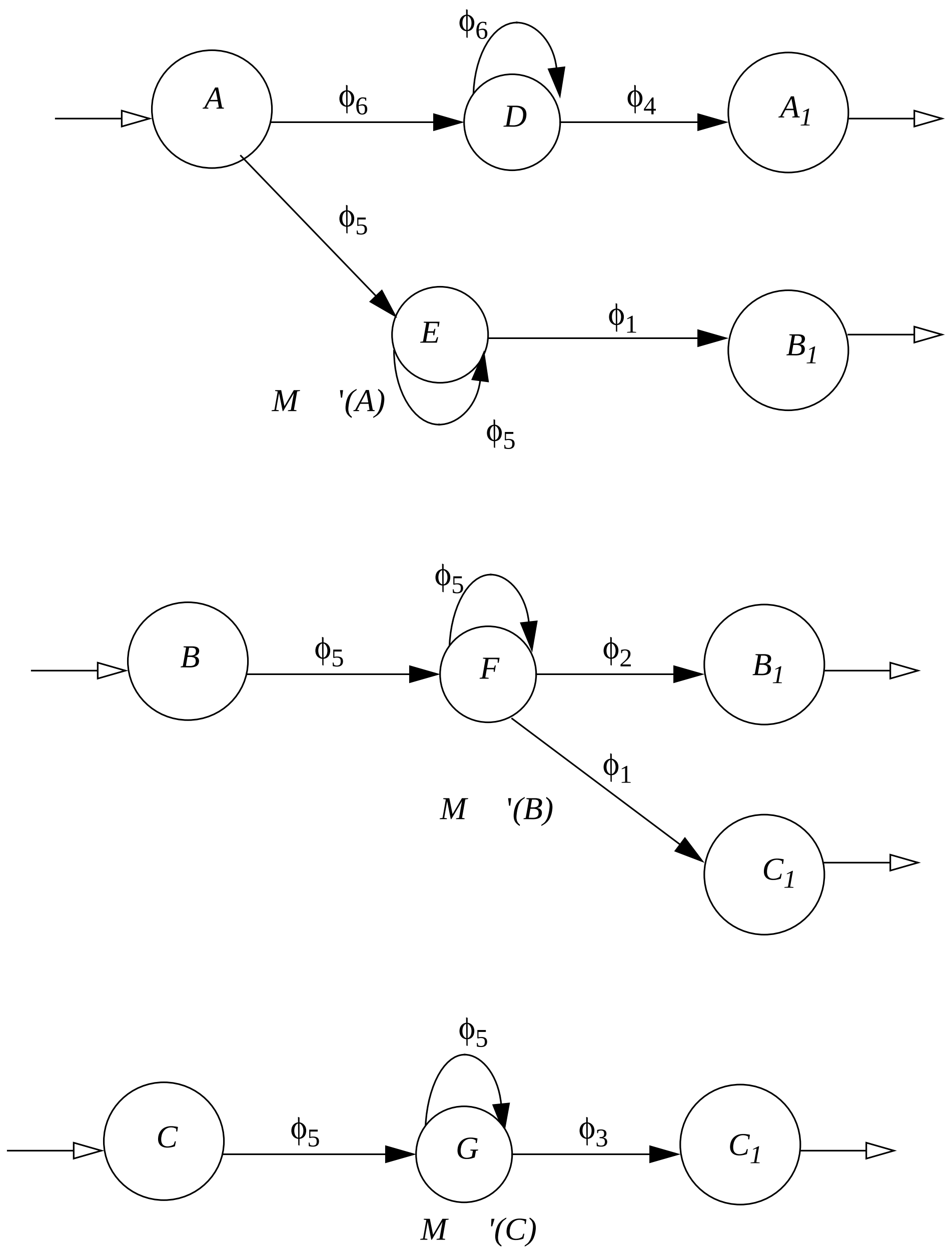

Figure 6.6 Three refining machines for the machine in Figure 6.3.

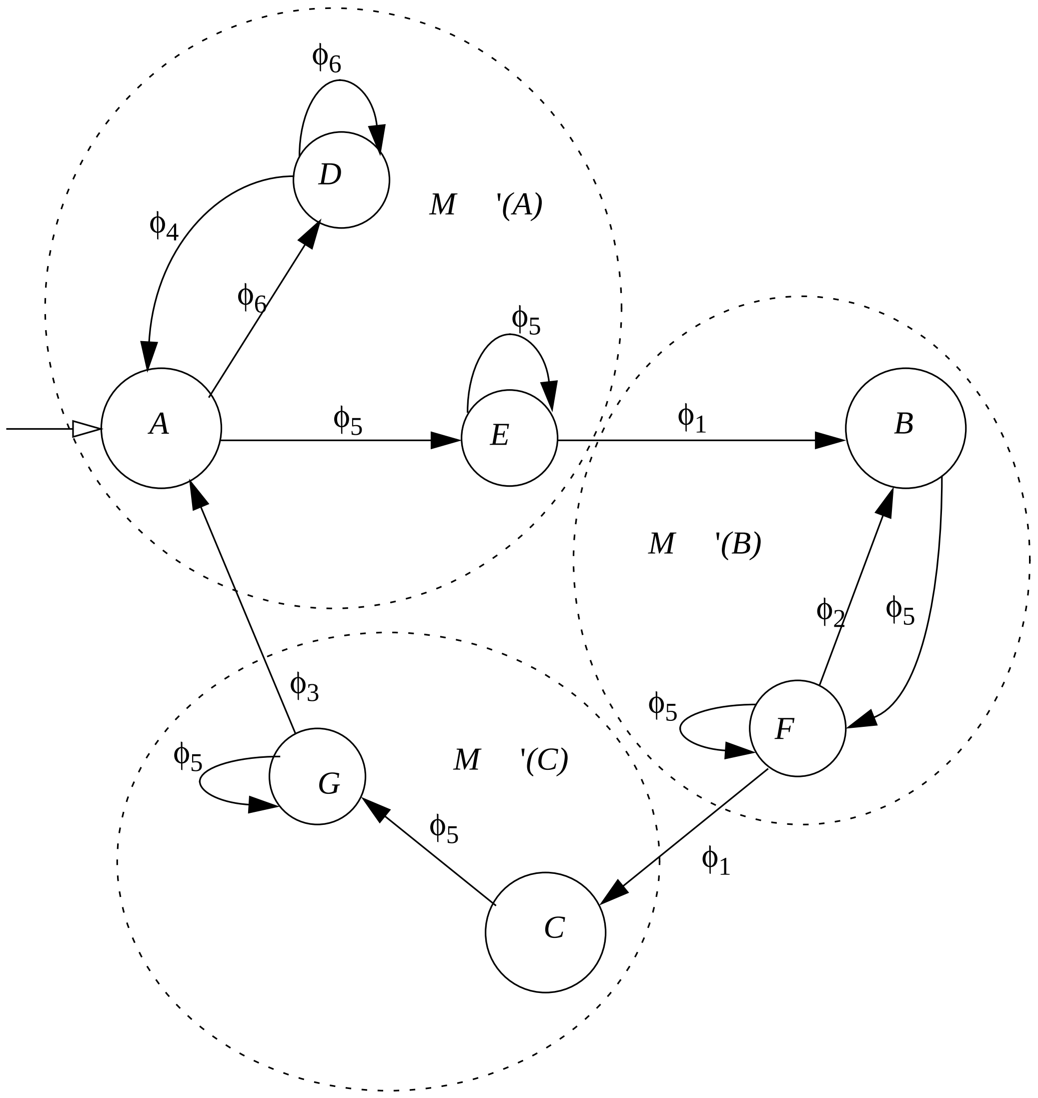

Figure 6.7 The refined machine from Figure 6.3.

Example 6.3.3.2.
We specify a simple computer program which enables users to “log in” to a computer system using their *username* and *password* and to replace their password. We require that each user can enter his/her password only once. As in Example 6.3.1.2, the keys used are: character keys, enter and backspace and each username or password will be entered as a sequence of character keys and backspace followed by enter. For simplicity we assume that all usernames and passwords will be words of at most 2 characters - the general case for n is similar. All usernames and passwords will be displayed along with appropriate messages.

The first stream X-machine model of the system may be found in Figure 6.7. The machine inputs will be all words of at most 2 characters, i.e.

Input = WORDS_2

An output will be a pair consisting of a message given by the system (i.e. “type in your password”, etc.) and the username or password displayed. Thus

Output = $\mathrm{WORDS}_2 \times \mathrm{MSGS}$

We need 5 messages thus:

MSGS = {*msg1, ..., msg5*}

A memory value will be a pair (*map*, *name*), where *map* will keep the map between the existing usernames and passwords for the existing accounts and *name* will store the username entered by the current user. Thus

Memory = $\mathrm{MAPS} \times \mathrm{WORDS}_2$,

where MAPS = $\mathrm{WORDS}_2 \rightarrow \mathrm{WORDS}_2$ is the set of partial functions from WORDS_2 to WORDS_2.

If map_0 is the current username/ password mapping then the initial memory value can be taken as $m_0 = (\mathrm{map}_0, `')$

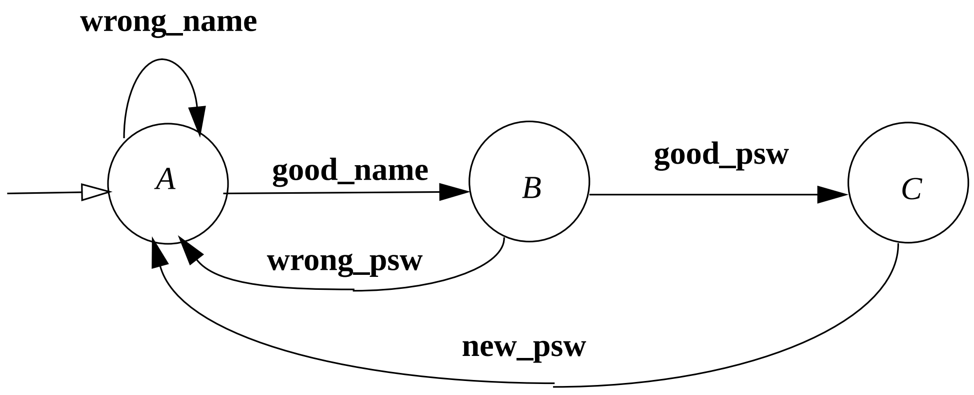

Figure 6.8 A simple stream X-machine for a computer “log in” system.

The 5 processing functions are defined as follows. We only show the effect of functions on their domain, they are undefined elsewhere.

Let *map* $\in$ MAPS, *name, str* $\in$ WORDS_2.

if *str* $\in$ domain *map* then **good_name**((*map*, *name*), *str*) = ((*msg1*, *str*), (*map*, *str*))

if *str* $\in$ (WORDS_2 - domain *map*) then

wrong_name((*map*, *name*), *str*) = ((*msg2*, *str*), (*map*, *name*))

if *name* $\in$ domain *map* and *map*(*name*) = *str* then

good_psw((*map*, *name*), *str*) = ((*msg3*, *str*), (*map*, *name*))

if ¬(*name* $\in$ domain *map* and *map*(*name*) = *str*) then
    wrong_psw((*map*, *name*), *str*) = ((*msg4*, *str*), (*map*, *name*))

new_psw((*map*, *name*), *str*) = ((*msg5*, *str*), (*map'*, *name*))
where *map'* = *map* $\oplus$ (*name* $\rightarrow$ *str*) and $\oplus$ denotes function overriding.

Up to this point we have not specified how the usernames and password are entered and displayed. This will be described by the refined machine along with the appropriate alphabet coverings.

The refined input and output alphabets are
    Input' = CHARS $\cup$ {*backspace*, *enter*} and
    Output' = CHARS $\cup$ {*del_ch*, *no_display*} $\cup$ MSGS

The input alphabet covering is u_2 with u_n as defined in Example 6.3.1.2. The output covering *v* is defined by:

    v(*msg*) = (‘’, *msg*), where *msg* $\in$ MSGS
    v(a* :: *no_display* :: *msg*) = *v*(a* :: *msg*)

$$v(a^* \; del\_ch :: msg) = \begin{cases} (‘a_1...a_{j-1}’, msg), \text{ if } v(a^* :: msg) = (‘a_1...a_j’, msg) \text{ with } j > 0 \\ (‘’, msg), \text{ if } v(a^* :: msg) = (‘’, msg) \end{cases}$$

where a* $\in$ seq(CHARS $\cup$ {*del_ch*, *no_display*})
    v(a* :: b :: *msg*) = ‘$a_1...a_j$ b’, where *v*(a* :: *msg*) = (‘$a_1...a_j$’, *msg*) with j $\geq$ 0
where a* $\in$ seq(CHARS $\cup$ {*del_ch*, *no_display*}), b $\in$ CHARS.

The state-transition diagrams of *M* '(*A*), *M* '(*B*) and *M* '(*C*) may be found in Figure 6.9.

    Memory' = *Memory* $\times$ $WORDS_2$
    m_0' = (m_0, < >)
h: *Memory* $\rightarrow$ *Memory'* is defined by
    h(*m*, < >) = *m*, *m* $\in$ *Memory*
We take *K* = *Q*, *L* = {*A1*, *B1*, *C1*} and r to be defined by:
    r(*A*) = *A1*, r(*B*) = *B1*, r(*C*) = *C1*.

The 9 processing functions are defined as follows. To avoid unnecessary detail, we only show the effect of functions on their domain.

Let *map* $\in$ MAPS, *name*, *str* $\in$ $WORDS_2$, *m* $\in$ *Memory*,

if *ch* $\in$ CHARS then
    type_ch1((*m*, *str*), *ch*) = (*ch*, (*m*, *str'*))
where
    str' = ‘$a_1...a_j$ *ch*’, where *str* = ‘$a_1...a_j$’ with j $\geq$ 0

if *ch* $\in$ CHARS then **type_ch2**((*m*, *str*), *ch*) = (*no_display*, (*m*, *str*))

press_bs1((*m*, *str*), *backspace*) = ((*del_ch*, (*m*, *str'*))
where *str'* = ‘$a_1 ... a_{j-1}$ ’, where *str* = ‘$a_1 ... a_j$’ with j > 0

press_bs2((*m*, *str*), *backspace*) = (*no_display*, (*m*, *str*))

if *str* $\in$ domain *map* then
good_name'(((*map*, *name*), *str*), *enter*) = (*msg1*, ((*map*, *str*), < >))

if *str* $\in$ ($WORDS_2$ - domain *map*) then
wrong_name'(((*map*, *name*), *str*), *enter*) = (*msg2*, ((*map*, *name*), < >))

if *name* $\in$ domain *map* and *map*(*name*) = *str* then
good_psw'(((*map*, *name*), *str*), *enter*) = (*msg3*, ((*map*, *name*), < >))

if ¬(*name* $\in$ domain *map* and *map*(*name*) = *str*) then
wrong_psw'(((*map*, *name*), *str*), *enter*) = (*msg4*, ((*map*, *name*), < >))

new_psw'(((*map*, *name*), *str*), *enter*) = (*msg5*, ((*map'* , *name*), < >))
where *map'* = *map* $\oplus$ (*name*, *str*)

The state-transition diagram of the refined machine may be found in Figure 6.10.

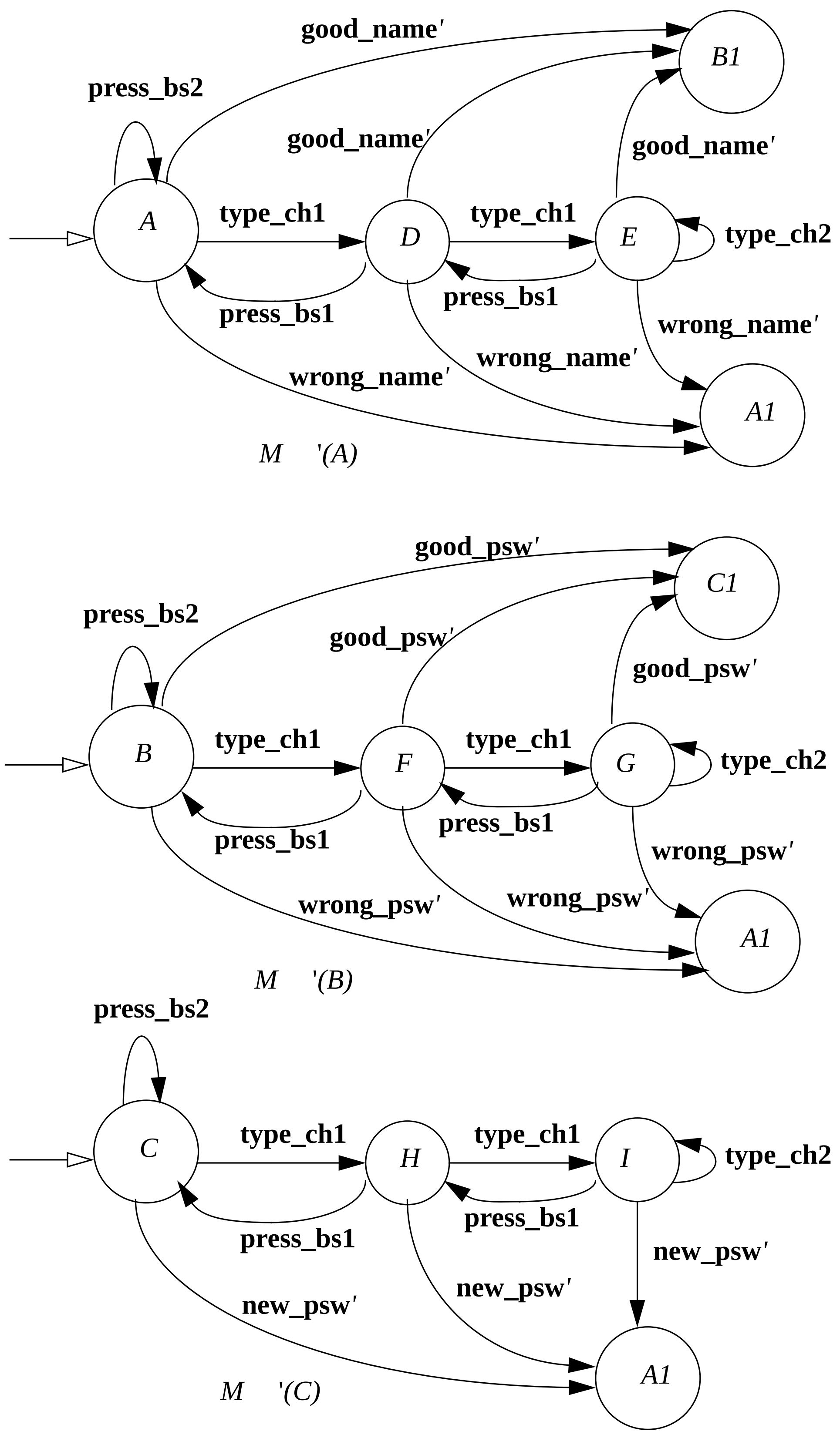

Figure 6.9 Refining machines for Figure 6.8.

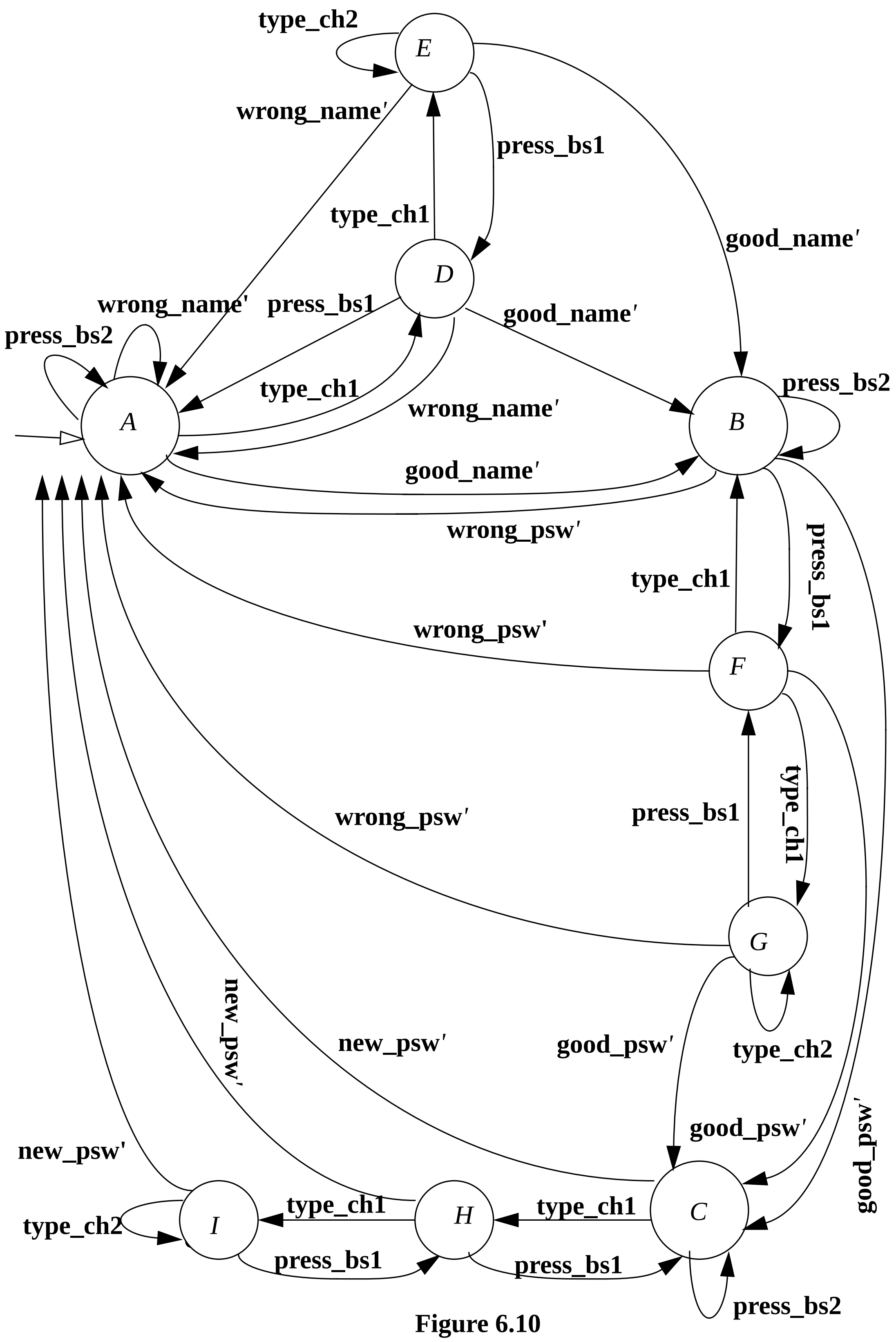

Figure 6.10

6.3.4. Equivalence of functional and state covering

The remaining part of this chapter will show that the two types of refinement presented are functionally equivalent. More precisely we shall prove that any state refinement of a stream X-machine M will also be functional refinement of M. Conversely, the behaviour of any functional refinement of M can be obtained by means of state refinement. In order to prove the first implication we state a preparatory result and introduce the concept of *structural refinement.*

Lemma 6.3.4.1.
Let M ' be a state refinement of M as above and let $s^{*\prime} \in$ domain u, $m_i' \in$ H. Then the following are true.
1. If p': r(q) $\rightarrow q_f'$ is a path in M ' with $(m_i', s^{*\prime}) \in$ domain $|p'|$ then $\exists\, q_f \in Q$ so that p': r(q) $\rightarrow$ d(q_f) is a path in M '(q).
2. For any path p' with $(m_i', s^{*\prime}) \in$ domain $|p'|$
p': r(q) $\rightarrow$ r(q_f) is a path in M ' $\Leftrightarrow$ p': r(q) $\rightarrow$ d(q_f) is a path in M '(q)

Proof:
1. If we assume otherwise then $\exists\, s_1^{*\prime}, s_2^{*\prime} \in$ seq(*Input*') with $s_1^{*\prime} :: s_2^{*\prime} = s^{*\prime}$ and p_1': r(q) $\rightarrow q_1'$ a path in M '(q) with $q_1' \in T_q$ and $(m_i', s_1^{*\prime}) \in$ domain $|p_1'|$. From requirements 2 and 5.ii of definition 6.3.3.1 it follows that $s_1^{*\prime} \in$ domain u. Since domain u is a prefix $s_1^{*\prime} = s^{*\prime}$ hence $p_1' = p'$ and $\exists\, q_f \in$ Q with $q_f' =$ d(q_f).
2. Follows from 1.

Definition 6.3.4.1.
Let M = (*Input*, *Output*, Q, *Memory*, Φ, **F**, q_0, m_0) and
M ' = (*Input*', *Output*', Q', *Memory*', Φ', **F**', q_0', m_0') be two stream X-machines so that seq(*Input*') covers seq(*Input*) w.r.t. u and seq(*Output*') covers seq(*Output*) w.r.t. v. Then we say that M ' is a *structural refinement* of M w.r.t. (u, v) if the following are true.
1. There exists an injective map r: $Q \rightarrow Q'$ from the state space of M to the state space of M ' whose image is K for which r(q_0) = q_0'.
2. There exists a (partial) surjective function h: *Memory'* $\rightarrow$ *Memory* whose domain is H for which h(m_0') = m_0.
3. For any $q_i' \in K$, $q_f' \in Q'$, $m_i' \in$ H, $m_f' \in$ *Memory'*, $s^{*\prime} \in$ seq(*Input*'), $g^{*\prime} \in$ seq(*Output*') and p': $q_i' \rightarrow q_f'$ a path in M ' with $|p'|(m_i', s^{*\prime}) = (g^{*\prime}, m_f')$ the following are true.
i) if $s^{*\prime} \in$ domain u then $q_f' \in K$, $m_f' \in$ H, $g^{*\prime} \in$ domain v
ii) if $s^{*\prime} \in$ seq$_+$(*Input*') - domain u^* then $g^{*\prime} \in$ seq$_+$(*Output*') - domain v^*.
4. For any $q_i \in Q$, $s \in$ *Input*, $m_i \in$ *Mattain*$_{qi}$(M), $m_i' \in$ h^{-1}(m_i) the following requirements are equivalent.
i) $\exists\, q_f \in Q$, $m_f \in$ *Memory*, $g \in$ *Output* and ϕ: $q_i \rightarrow q_f$ an arc in M such that $\phi(m_i, s) = (g, m_f)$.
ii) $\forall\, s^{*\prime} \in u^{-1}(s)\ \exists\, q_f' \in K$, $m_f' \in$ H, $g^{*\prime} \in$ domain v and p': r(q_i) $\rightarrow q_f'$ a path in M ' with $|p'|(m_i', s^{*\prime}) = (g^{*\prime}, m_f')$,
where q_f, q_f', m_f, m_f', g and $g^{*\prime}$ above satisfy r(q_f) = q_f', h(m_f') = m_f and $v(g^{*\prime}) = g$.

Lemma 6.3.4.2.

If M ' is a structural refinement of M w.r.t. (u, v) then M ' is a functional refinement of M w.r.t. (u, v).

Proof:
We prove the 2 requirements of lemma 6.3.2.1.
1. Let p' be a path in M ' that starts in q_0' with $\pi_1(|p'|(m_0', s^{*\prime})) = g^{*\prime}$.
If $s^{*\prime} \in$ (domain $u)^n$, by induction on n it follows that $g^{*\prime} \in$ (domain $v)^n$.
If $s^{*\prime} \in \text{seq}_+(Input') -$ domain u^* then from requirement 3.ii of definition 6.3.4.1 it follows that $g^{*\prime} \in \text{seq}_+(Output') -$ domain v^*.

2.
"i) $\Rightarrow$ ii)"
Let $s^* = s_1::...::s_n \in \text{seq}_+(Input)$, $g^* = g_1::...::g_n \in \text{seq}_+(Output)$ and p a path in M that starts in q_0 with $\pi_1(|p|(m_0, s^*)) = g^*$. Then $\exists\ m_1, ..., m_n \in Memory$, $\phi_1: q_0 \rightarrow q_1, ...,$ $\phi_n: q_{n-1} \rightarrow q_n$, n arcs in M with $\phi_1(m_0, s_1) = (g_1, m_1), ..., \phi_n(m_{n-1}, s_n) = (g_n, m_n)$. Then $\forall\ s_1^{*\prime} \in u^{-1}(s_1), ..., s_n^{*\prime} \in u^{-1}(s_n), m_1' \in h^{-1}(m_1) ,... , m_{n-1}' \in h^{-1}(m_{n-1})$ $\exists\ m_{1f}' \in h^{-1}(m_1), ..., m_{nf}' \in h^{-1}(m_n), g_1^{*\prime} \in v^{-1}(g_1), ..., g_n^{*\prime} \in v^{-1}(g_n)$ and $p_1': q_0' \rightarrow r(q_1), ..., p_n': r(q_{n-1}) \rightarrow r(q_n)$ n paths in M ' with $|p_1'|(m_0', s_1^{*\prime}) = (g_1^{*\prime}, m_{1f}'),$ $..., |p_n'|(m_{n-1}', s_n^{*\prime}) = (g_n^{*\prime}, m_{nf}')$. Thus $\forall\ s_1^{*\prime} \in u^{-1}(s_1), ..., s_n^{*\prime} \in u^{-1}(s_n)$ $\exists$ a path p' in M ' that starts in q_0' with $\pi_1(|p'|(m_0, s_1^{*\prime}::...::s_n^{*\prime})) = g_1^{*\prime}::...::g_n^{*\prime}$.

"ii) $\Rightarrow$ i)"
Let $s^* = s_1 ::... ::s_n \in \text{seq}_+(Input)$ and $s^{*\prime} = s_1^{*\prime}:: ...:: s_n^{*\prime}$ with $\forall$ j = 1...n, $s_j^{*\prime} \in u^{-1}(s_j)$ and let p' be a path in M ' that starts in q_0' with $\pi_1(|p'|(m_0', s^{*\prime})) = g^{*\prime}$, where $g^{*\prime} \in$ domain v^*. Then $\exists\ m_1', ..., m_n' \in Memory', g_1^{*\prime}, ..., g_n^{*\prime} \in \text{seq}(Output')$ and $p_1': q_0' \rightarrow q_1', ..., p_n': q_{n-1}' \rightarrow q_n'$ n paths in M ' with $|p_1'|(m_0', s_1^{*\prime}) = (g_1^{*\prime}, m_1'), ...,$ $|p_n'|(m_{n-1}', s_n^{*\prime}) = (g_n^{*\prime}, m_n')$, where $g^{*\prime} = g_1^{*\prime}::...::g_n^{*\prime}$. From requirement 3.i of definition 6.3.4.1 it follows that $\forall$ j = 1...n, $q_j' \in$ K, $m_j' \in$ H, and $g_j^{*\prime} \in$ domain v. Since $\forall$ j = 1...n, $s_j^{*\prime} \in u^{-1}(s_j)$ are arbitrary from requirement 4 of the same definition it follows that $\exists\ \phi_1: q_0 \rightarrow q_1, ..., \phi_n: q_{n-1} \rightarrow q_n$ n arcs in M with $\phi_1(m_0, s_1) = (g_1, m_1), ...,$ $\phi_n(m_{n-1}, s_n) = (g_n, m_n)$, where $\forall$ j = 1...n, $q_j' = r(q_j)$, $m_j = h(m_j')$ and $v(g_i^{*\prime}) = g_i$. Thus there exists a path p in M that starts in q_0 with $\pi_1(|p|(m_0, s_1 ::...:: s_n)) = g_1 ::...:: g_n$.

Lemma 6.3.4.3.
If M ' is a state refinement of M w.r.t. (u, v) then M ' is a structural refinement of M w.r.t. (u, v).

Proof:
Requirements 3.i and 4 of definition 6.3.4.1 follow from requirements 5.i and 6 respectively of definition 6.3.3.1 and lemma 6.3.4.1. The proof of requirement 3.ii is given next.
Let $q_i \in Q$, $s^{*\prime} \in \text{seq}_+(Input') -$ domain u^*, $m_i' \in$ H and p': $r(q_i) \rightarrow q_f'$ a path in M ' with $|p'|(m_i', s^{*\prime}) = (g^{*\prime}, m_f')$. Then either p': $r(q_i) \rightarrow q_f'$ is a path in M '(q) and hence $g^{*\prime} \in \text{seq}_+(Output') -$ domain v^* or there exist $s_1^{*\prime}, ..., s_n^{*\prime} \in$ domain u, $s_{n+1}^{*\prime} \in \text{seq}_+(Input') -$ domain u^*, $q_1, ..., q_n \in Q$, $m_1', ..., m_n' \in$ H, $p_1': r(q_i) \rightarrow r(q_1)$ a path in M '(q_i) with $|p_1'|(m_i', s_1^{*\prime}) = (g_1^{*\prime}, m_1'), ..., p_n': r(q_{n-1}) \rightarrow r(q_n)$ a path in

$M'(q_{n-1})$ with $|p_n'|(m_{n-1}', s_n*') = (g_n*', m_n')$ and $p_{n+1}': r(q_n) \rightarrow q_f'$ a path in $M'(q_n)$ with $|p_{n+1}'|(m_n', s_{n+1}*') = (g_{n+1}*', m_f')$. From requirements 5.i and 5.ii of definition 6.3.3.1 it follows that $g_1*', ..., g_n*' \in$ domain v, $g_{n+1}*' \in \mathrm{seq}_+(Output')$ - domain $v*$. Hence $g*' = g_1*' ::...:: g_{n+1}*' \in$ seq($Output'$) - domain $v*$.

Corollary 6.3.4.1.
If M' is a state refinement of M w.r.t. (u, v) then M' is a functional refinement of M w.r.t. (u, v).

Conversely, if M computes f and f' is a refinement of f then there is a simple refinement M' of M that computes f'. This result is proven next.

Theorem 6.3.4.1.
Let M be a stream X-machine, f the function it computes and let f' be a refinement of f w.r.t (u, v). Then there exists M' a state refinement of M w.r.t (u, v) so that M' computes f'.

Proof:
Let $M = (Input, Output, Q, Memory, \Phi, \mathbf{F}, q_0, m_0)$.
We take K = Q, r the identity, $Memory' = Memory \times \mathrm{seq}(Input') \times \mathrm{seq}(Input')$ and we define h: $Memory' \rightarrow Memory$ by
h$(m, x*', < >) = m$, where $m \in Memory$, $x*' \in$ domain $u*$

Then for any $q \in Q$ we construct an X-machine
$M'(q) = (Input', Output', R_q, Memory', \Phi_q', \mathbf{F}_q, q', m_q', T_q)$ as follows.

1. The initial state is q and T_q is as in definition 6.3.3.1.
2. $R_q = \{q\} \cup T_q$.
3. If $q = q_0$ then $m_q' = (m_0, < >, < >)$
4. $\Phi_q' = \{\phi_q'\} \cup \{\psi_{q\,\theta}' : \mathrm{d}(\theta) \in T_q, q \in Q\}$.

For $m \in Mattain_q(M)$ let y$(m) \in$ seq($Input'$) such that $\exists\, p: q_0 \rightarrow q$ with $\pi_2(|p|(m_0, u*(\mathrm{y}(m)))) = m$

Then ϕ_q' is defined by:

$\forall\, m \in Mattain_q(M)$, $x*' \in$ domain $u*$, $t*' \in$ seq($Input'$), $s' \in Input'$ with $t*' :: s' \notin$ domain u, $\phi_q'((m, x*', t*'), s') = (g', (m, x*', t*' :: s'))$
where

$$g' = \begin{cases} f'_{x*' ::\, t*'}(s'), \text{ if } \exists \text{ p}: q_0 \rightarrow q \text{ with } \pi_2(|p|(m_0, u*(x*'))) = m \\ f'_{\mathrm{y}(m)\, ::\, t*'}\ (s'), \text{ otherwise} \end{cases}$$

Also for any $\theta \in Q$, $\psi_{q\,\theta}'$ is defined by.

$\forall\, m \in Mattain_q(M)$, $x*' \in$ domain $u*$, $t*' \in$ seq($Input'$), $s' \in Input'$ with $t*' :: s' \in$ domain u and $\exists\, \phi: q \rightarrow \theta$ with $(m, u(t*' :: s')) \in$ domain ϕ

$\psi_{q\,\theta}'((m, x^{*'}, t^{*'}), s') = (g', (m_f, x^{*'}, t^{*'} :: s', < >))$,

where

$m_f = \pi_2(\phi(m, u^*(t^{*'} :: s')))$

$$g' = \begin{cases} f'_{x^{*'} :: t^{*'}}(s'), \text{ if } \exists\, p: q_0 \rightarrow q \text{ with } \pi_2(|p|(m_0, u^*(x^{*'}))) = m \\ f'_{y(m) :: t^{*'}}(s'), \text{ otherwise} \end{cases}$$

If the expressions attributed to g' in the above formulae are not defined then ϕ_q' or $\psi_{q\,\theta}'$ will be considered undefined for the corresponding values.

4. $\mathbf{F}_q$ is defined by:

$\mathbf{F}_q(q, \phi_q') = q$

$\mathbf{F}_q(q, \psi_{q\,\theta}') = d(\theta)$.

Using lemma 6.3.2.2 it is easy to show that $M'(q)$ satisfies requirements 5.i and 5.ii of definition 6.3.3.1. Also requirement 4 follows from lemma 6.3.2.3, the path p' that corresponds to an arc ϕ: $q \rightarrow \theta$ will be a sequence of ϕ_q's followed by a $\psi_{q\,\theta}'$.

It remains to show that M' constructed as in definition 6.3.3.2 computes f'. Let $s^{*'} \in$ seq($Input'$) and let n ≥ 1 the largest natural number so that $s^{*'}$ can be written as $s^{*'} = s_1^{*'} ::...:: s_n^{*'}$ with $s_1^{*'}, ..., s_{n-1}^{*'} \in$ domain u and $s_n^{*'} \in$ seq($Input'$) - domain u^* (if $s^{*'} \in$ domain u^* then we take $s_n^{*'} = < >$). Then by induction on the length of $s^{*'}$ it follows that:

1. If p': $q_0 \rightarrow q$ is a path in M' with $\pi_2(|p'|((m_0, < >, < >), s^{*'}) = (m, x^{*'}, t^{*'})$ then $x^{*'} = s_1^{*'}::...::s_{n-1}^{*'}$ and $t^{*'} = s_n^{*'}$.

Also by induction on the length of $s^{*'}$, using lemma 6.3.2.3, it follows that

2. $f'(s^{*'}) = g^{*'} \Leftrightarrow \exists\, q \in Q, m \in Memory$ and a path p': $q_0 \rightarrow q$ in M' with $|p'|((m_0, < >, < >), s^{*'}) = ((m, s_1^{*'}::...::s_{n-1}^{*'}, s_n^{*'}), g^{*'})$

Hence M' computes f'.

Corollary 6.3.4.2. (*The fundamental theorem of refinement.)*
Let M be a stream X-machine, f the (partial) function it computes and let f' be a (partial) stream function. Then f' is a refinement of f w.r.t (u, v) if and only if there exists M' a state refinement of M w.r.t. (u, v) that computes f'.