maker4\infmatic:Version 1.3.

Another look at computability

Florentin Ipate and Mike Holcombe,
Fomal Methods and Software Engineering Research Group (Formsoft),
Department of Computer Science,
University of Sheffield, England,U.K.

Correspondence to: M. Holcombe or F. Ipate, Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, England, U.K. Email: M.Holcombe@dcs.shef.ac.uk or F. Ipate@dcs.shef.ac.uk

Abstract. The theory of computable functions is well known and has given rise to many classes of computational models of varying power and usefullness. We take another look at this subject using the idea of a generalised machine - the X-machine - to provide some further insights into the issue and to discuss an elegant general approach to the question of classifying computational models including some of the so-called "Super-Turing" models.

This paper investigates a number of classes of X-machines. It considers their relative computational capabilities and contrasts these with other important models. It is shown that a certain class of these machines - the 2-stack straight move stream X-machine - computes precisely the class of partial recursive functions.

The importance of this work to the theory of testing of systems is stressed.

1. Introduction.

The Turing model [1], [2], has been a cornerstone of the theory of computation for many years and the Chomsky hierarchy of machines a useful mechanism for categorising machines and languages of different capabilities. The theory of X-machines [1], [3], however, offers an alternative approach which has proved valuable and enlightening. For most purposes the Turing model is too restrictive and low-level for serious application as a vehicle for the description and analysis of computational devices. Howevere, the X-machine model offers several important benefits over the Turing model:

1. it is a convenient abstraction that enables different classes of machines to be defined in terms of classes of simple transition functions, thus providing a more unified and coherent approach to the machine hierarchy problem;
2. the data abstraction capabilities of the model make it feasible for use as a basic universal specification and analysis language [3];
3. the approach provides a general framework for the discussion of computational models of greater generality than the Turing model, for example continuous, highly parallel algorithms which are "super-Turing" as in [4];
4. the precision with which the model defines what is practically implementable with current technology allows for the discussion of formal refinement processes and for the verification that implementations satisfy their behavoural requirements as expressed in terms of X-machine specifications, [5].

This has some important application potential, also, in the theory of system and software testing where assumptions have to be made about the form of an implementation which must depend on the existence of a feasible computational model as a basis for the representation of this implementation. Previous approaches required the assumption that the implementation was a machine of a particular type, for example a finite state machine, before the question of whether the implementation met its specification could be discussed. Using the X-machine model, such approaches can be enhanced and generalised [6], [7].

We demonstrate some important properties of a natural class of X-machines in terms of their computational capabilities. Thus, for example, the natural classes of X-machines that model computational systems that process a stream of input data into a stream of output data are defined and examined. These, so called, *stream X-machines*, *generalised stream X-machines* and *straight-move stream X-machines* have a number of important properties which relate to other approaches to the modelling of computation.

The principal result is that:

- *2-stack straight move stream X-machines* compute precisely the class of *partial recursive functions*;

<u>2. X-machines - a general computational model.</u>

We begin with a number of essential definitions and notational matters.

<u>*2.1 The X-machine model*</u>

The original definition of the X-machine is due to Samuel Eilenberg, [1], where it was presented as an alternative to the Finite State Machine, Pushdown Machine, Turing Machine and other standard types of machine. The theory was not developed to any great extent in this source, however. Here we present the definition of an X-machine in its most general form - although not all the features are needed for our purposes.

Definition 2.1.1:
An *X-Machine* is a 10-tuple $M = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T)$, 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 relations* respectively, used to convert the input and the output sets into, and from, the fundamental set, i.e.

$\alpha: Y \leftrightarrow X, \quad \beta: X \leftrightarrow Z$

4. Q is the (finite) *set of states*.
5. Φ is the *type* of *M*, a set of relations on X, i.e.

$\Phi: P\,(X \leftrightarrow X)$

The type of the machine is the class of relations (usually partial functions) that constitute the elementary operations that the machine is capable of performing. *P* S denotes the power set of S. 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*.

$F: Q \rightarrow (\Phi \rightarrow P\,Q)$

So, for state $q \in Q$, $F(q): \Phi \rightarrow P\,Q$ is a partial function.
However, when it is convenient, F can be treated like a partial function with two arguments, i.e. $F(q, \varphi) = (F(q))(\varphi)$. F is often described by means of a state-transition diagram.
7. I and T are the sets of *initial* and *terminal 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 relations or 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.

Definition 2.1.2:
If $q, q' \in Q$, $\varphi \in \Phi$ and $q' \in F(q, \varphi)$, we say that φ is the *arc* from q to q', represented thus:

$$\begin{tikzcd} q \arrow[r, "\phi"] & q' \end{tikzcd}$$

Definition 2.1.3:
If $q, q' \in Q$ are such that there exist $q_1,..., q_n \in Q$ and $\varphi_1,..., \varphi_{n+1} \in \Phi$ with

$$\begin{tikzcd} q \arrow[r, "\phi_1"] & q_1 \arrow[r, "\phi_2"] & q_2 \cdots & q_n. \arrow[r, "\phi_{n+1}"] & q' \end{tikzcd}$$

then $(\langle q, q_1,..., q_n, q'\rangle, \langle\varphi_1,..., \varphi_{n+1}\rangle)$ is the *path* from q to q'. Each path c is labelled with |c|, where

$|c| = \varphi_1... \varphi_{n+1}:X\rightarrow X$

is the relation computed by the machine when it follows that path.
When the state sequence is not relevant we shall refer to a path as the sequence of relations, i.e. $c = \varphi_1... \varphi_{n+1}$.
A *successful path* is one that starts in an initial state (in I) and ends in a final one (from T).
A *loop* is a path whose initial state is also terminal (i.e. a path from a state to itself).

Definition 2.1.4:
The *behaviour* of *M* is the relation

$|M|: X\rightarrow X$

defined as

$|M| = \cup |c|$

with the union extending over all the successful paths c in *M* .

Given $y \in Y$, the operations of the X-machine *M* on Y consist of:
1. Picking a path c, from a start state q_i ($q_i \in I$), to a final state q_t ($q_t \in T$) i.e.

$|c|: q_i \rightarrow q_t$

2. (Optional) Applying α to the input to convert it to the internal type X.
3. Applying |c|, if it is defined for α(y). Otherwise, go back to step 1.
4. (Optional) Applying β to get the output.
Therefore, the operation can be summarised as β(|c|(α(y))). Note that the output may be non-deterministic or produce a set of outputs from a given input.

Definition 2.1.5:
The composite relation $f_{\mathcal{M}}$ given by:

$$\alpha \circ |\mathcal{M}| \circ \beta : Y \to Z, \text{ ie.}$$

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

is called the *relation computed* by $\mathcal{M}$.

If $\mathcal{M}$ is a X-machine acceptor (i.e. $\Gamma = \emptyset$), then the relation (partial function) f computed by it will have only one output value, i.e.

$$f(x) = \begin{cases} c, \text{ if } x \in \text{dom } f \\ \emptyset, \text{ otherwise} \end{cases}$$

where c is an arbitrary constant.
We call L = dom f the *language accepted by the machine* $\mathcal{M}$ *.*

Now suppose that X is a fixed data set and Φ, Φ' are types of relations on X such that $\Phi \subseteq \Phi$ then the class of relations computed by X-machines of type Φ will be contained in the class of relations computed by X-machines of type Φ'. Thus it is conceivable that the study of the relations computed by X-machines of different structures and types provides a mechanism for classifying a wide range computable relations in a convenient way. For example, we would be interested in defining a class of partial functions that are computable by one class of X-machines and then to use this class of partial functions as the elements of the sets Φ that define a more general class of, say, X'-machines. By carefully defining the classes of machines and the corresponding partial functions computed by them we have the structure for a framework for discussing many types of computational model that extend far beyond traditional Turing-based models.

There are a number of special classes of X-machines that are of interest here. We discuss the relative computational power of these classes of machines both from the point of view of what relations they can compute but also what sort of closure properties they possess - for example if the functions or relations of the type Φ satisfy some property is this property shared by the functions or relations computed by the machine?

Before proceeding any further we shall describe a general class of X-machines to which we shall be referring in this paper.

We let $Y = \Sigma^*$, $Z = \Gamma^*$, where Σ (*input alphabet*) and Γ (*output alphabet*) are finite alphabets. Thus relations $f: \Sigma^* \to \Gamma^*$ will be computed. The set X will have the form

$$X = \Gamma^* \times M \times \Sigma^*,$$

where M is a monoid called *memory.*
In practice M will usually be a product $\Omega_1^* \times ... \times \Omega_r^*$, where $\Omega_1, ... \Omega_r$ are finite alphabets: in this case we shall say that X has r+2 registers of which one (the last one) is the input register, one (the first one) is the output register and the intermediate r registers are memory registers.
For $x \in X$, $x = (g, m, s)$ the values of output register, input register and memory will be referred to as s = In(x), g = Out(x), m = Mem(x) respectively. Therefore, we have $Out: X \to \Gamma^*$, $In: X \to \Sigma^*$, $Mem: X \to M$, $Out(g, m, s) = g$, $In(g, m, s) = s$, $Mem(g, m, s) = m \ \forall g \in \Gamma^*, s \in \Sigma^*, m \in M$.

This model is sufficiently general to model many common types of machine from finite state machines (where the memory is trivial) to Turing machines (where, as we shall see, the memory is a model of the tape), see [1].

2.2. Deterministic X-machines

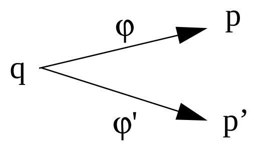
The aim of defining deterministic machines is to compute partial functions rather than relations. In a deterministic X-machine, there is at most one possible transition for any state q and any $x \in X$.

Definition 2.2.1:
An X-machine M is called *deterministic* if:

1. α and β are partial functions, not relations:
   $\alpha: Y \rightarrow X$, $\beta: X \rightarrow Z$
2. Φ contains only partial functions on X rather than relations:
   $\Phi: P\,(X \rightarrow X)$
3. F maps each pair $(q, \varphi) \in Q \times \Phi$ onto at most a single next state:
   $F: Q \rightarrow (\Phi \rightarrow Q)$

A partial function is used because every $\varphi \in \Phi$ will not necessarily be defined as the label to an edge in every state.

4. I contains only one element (i.e. $I = \{q_0\}$, where $q_0 \in Q$)
5. If φ and φ' are distinct edges emerging from the same state then dom $\varphi \cap$ dom φ' =



$\varnothing$

If we consider F as a function with two arguments, the condition above can be written as:
$\forall\, q \in Q, \varphi, \varphi' \in \Phi$, if $(q, \varphi), (q, \varphi') \in$ dom F then dom $\varphi \cap$ dom $\varphi' = \varnothing$.

These conditions will frequently (but not always) ensure $f = \alpha \circ |M| \circ \beta$ is a partial function. However, if a deterministic X-machine satisfies an additional condition, it will compute a partial function.

Definition 2.2.2:
A path $|c| = \varphi_1 \ldots \varphi_{n+1}: X \rightarrow X$ is called *trivial*, if $\exists\, x \in$ dom $|c|$ such that $\mathrm{In}(|c|(x)) = \mathrm{In}(x)$. In other words, a trivial path is one along which the machine does not change the value of the input register for some values of X, while possibly changing the output and memory registers.

Then, we have the following straightforward result:

Proposition 2.2.3 [1]:
If M is a deterministic stream X-machine in which no non-trivial path connects two terminal states, then M computes a partial function.

Before defining the two X-machine models that we shall be concentrating on in this paper, 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 Σ an alphabet and let $s \in \Sigma^*$. Then we define the functions
$L_s, R_s : \Sigma^* \to \Sigma^*$ by

$$L_s(x) = sx, \; R_s(x) = xs \; \forall x \in \Sigma^*$$

and the partial functions
$L_{-s}, R_{-s} : \Sigma^* \to \Sigma^*$ by

$$L_{-s}(x) = s^{-1}x \text{ (i.e. dom } L_{-s} = \{s\}\Sigma^* \text{ and } L_{-s}(sx) = x \; \forall x \in \text{dom } L_{-s})$$

$$R_{-s}(x) = xs^{-1} \text{ (i.e. dom } R_{-s} = \Sigma^*\{s\} \text{ and } R_{-s}(xs) = x \; \forall x \in \text{dom } R_{-s})$$

Note: Here sx is s concatenated to x (or in some notations s::x).
Obviously, $L_s L_t = L_{ts}$, $R_s R_t = R_{st}$, $L_{-s} L_{-t} = L_{-ts}$, $R_{-s} R_{-t} = R_{-st}$ $\forall s, t \in \Sigma^*$.
We shall denote by $I: \Sigma^* \to \Sigma^*$ the identity function.

3. Stream X-machines and their generalisations.

A number of important classes of X-machines have been identified and studied. Typically the classes are defined by restrictions on the underlying data set X and the type Φ of the machines. We introduce three such classes.

3.1. (Generalised) stream X-machines; straight move stream X-machines

The main type of X-machines that we consider here are those that process their input streams 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.

Definition 3.1.1:
Let Σ and Γ two finite alphabets, and let $\delta \notin \Sigma \cup \Gamma$ (we call δ the blank or end marker), $\Sigma' = \Sigma \cup \{\delta\}$, $\Gamma' = \Gamma \cup \{\delta\}$.
Then, an X-machine $\mathcal{M} = (Q, \Sigma, \Gamma, M, \alpha, \beta, \Phi, F, Q_0, T, m_0)$ with
$X = \Gamma'^* \times M \times \Sigma'^*$, is called a *stream X-machine* if:

1. The type is defined as
$\Phi = \{R_\gamma \mid \gamma \in \Gamma\} \times \Phi_M \times \{L_{-\sigma} \mid \sigma \in \Sigma\} \cup \{R_\delta\} \times \Phi_M \times \{L_{-\delta}\}$,
where $\Phi_M = \{\phi \mid \phi: M \leftrightarrow M\}$ is a set of relations on M.

Therefore, each transition function must remove the head of the input stream and add an element to the rear of the output stream, and, furthermore, no transition is allowed to use information from the tail of the input or any of the output. Additionally, whenever the end marker δ is processed, the output is also δ.

2. The input and output codes
$\alpha: \Sigma^* \to X$, $\beta: X \to \Gamma^*$ are defined by
$\alpha(s) = (1, m_0, R_\delta(s)) \; \forall s \in \Sigma^*$,

$$\beta(g, m, s) = \begin{cases} R_{-\delta}(g), & \text{if } s = 1; \\ \emptyset, & \text{otherwise} \end{cases}$$

$\forall g \in \Gamma^*$, where $m_0 \in M$ is called the initial value of the memory and 1 is the empty string.

For any input string $s \in \Sigma^*$ and any corresponding successful path $|c|$, the computation will be :

$$s \xrightarrow{\alpha} s\delta \xrightarrow{|c|} g\delta \xrightarrow{\beta} g$$

with $g \in \Gamma^*$. Therefore the partial function computed by the machine $f = \alpha \circ |M| \circ \beta$ has type $f: \Sigma^* \to \Gamma^*$ (with no occurrence of δ in the alphabets). The end marker could be totally eliminated and α and β could be defined as:

$\alpha(s) = (1, m_o, s)\ \forall\ s \in \Sigma^*$,

$\beta(g, m, s) = g$, if $s = 1$

$= \varnothing$, otherwise

A stream X-machine without end marker could be easily transformed into one with end marker, but not vice versa.

If the type is $\Phi = \{R_g \mid g \in \Gamma^*\} \times \Phi_M \times \{L_{-\sigma} \mid \sigma \in \Sigma\} \cup \{R_\delta\} \times \Phi_M \times \{L_{-\delta}\}$ then the X-machine is called a *generalised stream X-machine*. Therefore, the output at each state transition can be a string from Γ^* rather then just a symbol from Γ.

Empty input moves (i.e. moves where the input string remains unchanged) are not allowed in (generalised) stream X-machines. If we accept such moves, we get a more general X-machine model: -

Definition 3.1.2:
Let $\Phi_M = \{\phi \mid \phi: M \leftrightarrow M\}$ be a set of relations on M and consider the following extensions to the original definition:

1. we expand the type Φ by considering $\Phi = \Phi_1 \cup \Phi_2 \cup \Phi_3 \cup \Phi_4$ where:
$\Phi_1 = \{R_\gamma \mid \gamma \in \Gamma \cup \{\delta\}\} \times \Phi_M \times \{L_{-\sigma} \mid \sigma \in \Sigma \cup \{\delta\}\}$, i.e. $\varphi \in \Phi_1$ reads the head of the input stream (possibly δ) and adds an output character (possibly δ) to the end of the output string.
$\Phi_2 = \{I\} \times \Phi_M \times \{L_{-\sigma} \mid \sigma \in \Sigma \cup \{\delta\}\}$, i.e. $\varphi \in \Phi_2$ reads the head of the input stream (possibly δ) and leaves the output string unchanged.
$\Phi_3 = \{R_\gamma \mid \gamma \in \Gamma \cup \{\delta\}\} \times \Phi_M \times \{I\}$, i.e. $\varphi \in \Phi_3$ leaves the input string unchanged while adding an output character (possibly δ) to the end of the output string.
$\Phi_4 = \{I\} \times \Phi_M \times \{I\}$, i.e. $\varphi \in \Phi_4$ leaves both the input and output strings unchanged.

2. We further assume that $\forall\ q \in T, \forall\ \varphi \in \Phi_3 \cup \Phi_4$ then
$(q, \varphi) \notin$ dom F.
(Therefore, no empty input transition is allowed from a terminal state.)

3. Any path p from an initial state to a terminal one (i.e. from q to q' where $q \in Q_o$, $q' \in T$) has the form $p = \varphi_1 \ldots \varphi_i \ldots \varphi_j \ldots \varphi_n$, for some $i \le j$, where
   a. if $i < j$, $\varphi_i \in (\{R_\gamma \mid \gamma \in \Gamma\} \cup \{I\}) \times \Phi_M \times \{L_{-\delta}\}$,
   $\varphi_j \in \{R_\delta\} \times \Phi_M \times (\{L_{-\sigma} \mid \sigma \in \Sigma\} \cup \{I\})$.
   if $i = j$, then $\varphi_i \in \{R_\delta\} \times \Phi_M \times \{L_{-\delta}\}$.

   b. for $k \in \{1, \ldots, j\text{-}1\} - \{i\}$,

$\varphi_k \in (\{R_\gamma \mid \gamma \in \Gamma\} \cup \{I\}) \times \Phi_M \times (\{L_{-\sigma} \mid \sigma \in \Sigma\} \cup \{I\})$.

c. for $k \in \{j+1, ..., n\}$, $\varphi_k \in \{I\} \times \Phi_M \times (\{L_{-\sigma} \mid \sigma \in \Sigma\} \cup \{I\})$.

In other words, for any path which starts in an initial state and ends in a terminal one, the machine reads only one blank (δ) and produces only one blank (δ) in this order. Furthermore, no other outputs are produced after the blank (δ) is written at the end the output string.

Then the tuple $M = (Q, \Sigma, \Gamma, M, \alpha, \beta, \Phi, F, Q_o, T, m_o)$, where α and β are defined as above, is called a *straight-move stream X-machine*.

Φ_1 is called the set of *non-empty input and non-empty output operations*.
Φ_2 is called the set of *non-empty input and empty output operations*.
Φ_3 is called the set of *empty input and non-empty output operations*.
Φ_4 is called the set of *empty input and empty output operations*.

A straight-move stream X-machine can process the empty string 1 and can produce 1 as the output. It is fairly clear that a straight move stream X-machine computes a relation $f: \Sigma^* \to \Gamma^*$, where $f = \alpha \circ |M| \circ \beta$.
Note: The empty string 1 should not be confused with the end marker δ.

Obviously, the straight move stream X-machine model is more general than the generalised stream X-machine one. Conversely, the generalised stream X-machine is a more efficient or a faster version of the straight move stream X-machine. No empty input moves are allowed in a generalised stream X-machine and the machine reads a character every time a move is performed. Even more importantly, the generalised stream X-machine model ensures that for any input string the machine will stop its computation in a finite time, therefore avoiding the 'halting problem'. For an input string s of length n, the machine will give the result f(s) in at most n+1 moves. We formalise this idea in what follows.

Definition 3.1.3:
The type Φ is called *fully computable* if $\forall \varphi \in \Phi$, then there exists an algorithm A such that A computes φ and $\forall x \in X$, x will cause A to stop in a finite time (i.e. $\neg \exists x \in X$ such that A will run forever while computing $\varphi(x)$).

Note: We consider an algorithm as being a procedure involving a finite number of basic operations. This notion is in some way ambiguous since it is dependent on the basic operations allowed. This can be addressed, according to Church's thesis, by considering the Turing machine as the general model for an algorithm. Then, for a deterministic X-machine (this is the case we shall be addressing in this paper) our definition of full computability becomes:
the type Φ is called *fully computable* if $\forall \varphi \in \Phi$, φ is a partial recursive function and dom φ is a recursive set.
However, the more general definition above is sufficient for our purpose at the moment.

Proposition 3.1.4:
Let $M = (Q, \Sigma, \Gamma, M, \alpha, \beta, \Phi, F, Q_o, T, m_o)$ be a generalised stream X-machine with Φ fully computable. Then, the relation $f: \Sigma^* \to \Gamma^*$ computed by M is fully comput-

able. Hence, the class of fully computable relations is closed under the generalised stream X-machine operator.

Proof:
We define k = card Φ' where Φ' is the subset of Φ used by M. Let $x = \sigma_1 \dots \sigma_n$, with $\sigma_1, \dots, \sigma_n \in \Sigma$. Hence $\alpha(x) = (1, m_o, \sigma_1 \dots \sigma_n\delta)$. The number of paths determined through the machine by x is $N \leq k^{n+1}$. Therefore, $\varphi(x)$ is determined by applying at most $N(n+1) \leq (n+1)k^{n+1}$ algorithms (i.e. an algorithm for each φ which processes σ_i or δ).
Hence $f = \alpha \circ |M| \circ \beta$ is fully computable. ◊

Obviously, the proposition above is not true for a straight move stream X-machine since it can contain loops formed only by empty input operations and the machine can run forever following such loops.

3.2. Deterministic (straight move) stream X-machines

From definition 2.2.1, it follows that a deterministic generalised stream X-machine is one which has only one transition for a certain triplet $(q, m, \sigma) \in Q \times M \times \Sigma'$,
i.e. $\forall (q, m, \sigma) \in Q \times M \times \Sigma'$ and $\varphi, \varphi' \in \Phi$, if $\Gamma'^* \times \{m\} \times \{\sigma\}\Sigma'^* \cap \text{dom } \varphi \neq \emptyset$ and $\Gamma'^* \times \{m\} \times \{\sigma\}\Sigma'^* \cap \text{dom } \varphi' \neq \emptyset$, then either $(q, \varphi) \notin \text{dom } F$ or $(q, \varphi') \notin \text{dom } F$. Here Φ is a set of partial functions.

A *deterministic straight move stream X-machine* will satisfy the following:

1. there is only one possible transition for any triplet $(q, m, \sigma) \in Q \times M \times \Sigma'$, i.e. $\forall (q, m, \sigma) \in Q \times M \times \Sigma'$ and $\varphi, \varphi' \in \Phi_1 \cup \Phi_2$, if $\Gamma'^* \times \{m\} \times \{\sigma\}\Sigma'^* \cap \text{dom } \varphi \neq \emptyset$ and $\Gamma'^* \times \{m\} \times \{\sigma\}\Sigma'^* \cap \text{dom } \varphi' \neq \emptyset$, then either $(q, \varphi) \notin \text{dom } F$ or $(q, \varphi') \notin \text{dom } F$.

2. there is no pair $(q, m) \in Q \times M$ where both a letter $\sigma \in \Sigma'$ and the empty input string 1 can be read, i.e. $\forall (q, m, \sigma) \in Q \times M \times \Sigma'$ and $\varphi \in \Phi_1 \cup \Phi_2$, $\varphi' \in \Phi_3 \cup \Phi_4$, if $\Gamma'^* \times \{m\} \times \{\sigma\}\Sigma'^* \cap \text{dom } \varphi \neq \emptyset$ and $\Gamma'^* \times \{m\} \times \Sigma'^* \cap \text{dom } \varphi' \neq \emptyset$ then $(q, \varphi) \notin \text{dom } F$ or $(q, \varphi') \notin \text{dom } F$.

3. there is no pair $(q, m) \in Q \times M$ where two different empty input transitions are allowed, i.e. $\forall (q, m) \in Q \times M$ and $\varphi, \varphi' \in \Phi_3 \cup \Phi_4$, if $\Gamma'^* \times \{m\} \times \Sigma' \cap \text{dom } \varphi \neq \emptyset$ and $\Gamma'^* \times \{m\} \times \Sigma'^* \cap \text{dom } \varphi' \neq \emptyset$, then $(q, \varphi) \notin \text{dom } F$ or $(q, \varphi) \notin \text{dom } F$.

Obviously, no trivial paths exist in a *stream X-machine* or a *generalised stream X-machine* and no trivial paths start from a terminal state in a deterministic *straight move stream X-machine*. Hence:

Proposition 3.2.1:
Any deterministic stream X-machine, generalised stream X-machine or straight move stream X-machine computes a partial function.

However, stream X-machines and generalised stream X-machines compute special classes of (partial) functions, as we shall see in the following subsection.

3.3. Stream functions; generalised stream functions

Each class of machine defines a type of function or relation between the input alphabet and the output alphabet. If there are natural mathematical descriptions of these functions this will be an indication of the extent to which the notions introduced in this paper are natural ones.

Definition 3.3.1:
Let $f: \Sigma^* \to \Gamma^*$ be a partial function. Then f is called *segment preserving* if:
$\forall$ s, t $\in \Sigma^*$, if s, st $\in$ dom f then $\exists$ u $\in \Gamma^*$ such that f(st) = f(s)u.

Definition 3.3.2:
Let $f: \Sigma^* \to \Gamma^*$ be a partial function. If $|f(s)| = |s|$ $\forall$ s $\in$ dom f then f is called *length preserving*.
Note: |s| denotes the length of the string s.

Definition 3.3.3:
Let $f: \Sigma^* \to \Gamma^*$ be a partial function. Then f is called a *partial stream function* if f is both segment preserving and length preserving.

If we replace the length preserving condition by a Lipschitz type condition, we get the definition of a partial generalised stream function;

Definition 3.3.4:
Let $f: \Sigma^* \to \Gamma^*$ be a segment preserving partial function. Then f is called a *partial generalised stream function* if:
$\exists$ k $\in$ N such that $\forall$ s, t $\in \Sigma^*$, if s, st $\in$ dom f, then $||f(st)| - |f(s)|| \le k|t|$.

Definition 3.3.5:
A partial (generalised) stream function $f: \Sigma^* \to \Gamma^*$ is *complete* if:
$\forall$ s, t $\in \Sigma^*$, if st $\in$ dom f, then s $\in$ dom f.

We have the following characterisation of deterministic (generalised) stream X-machines:

Proposition 3.3.6:
1. Any deterministic *stream X-machine* computes a *partial stream function*.
2. Any deterministic *generalised stream X-machine* computes a *partial generalised stream function*.
3. Any *stream X-machine* (*generalised stream X-machine*) without end marker and with all the states terminal (T = Q) computes a complete *partial stream function* (*partial generalised stream function*).

Proof:
By induction on t it follows that f(st) = f(s)u $\forall$ s, t $\in$ dom f .
2, 3. If the machine is a generalised stream X-machine, then we take
$k = \max \{|g| \mid (R_g, \phi, L_{-\sigma}) \in \Phi'\}$, where Φ' is the finite set of Φ used to label the arcs of the machine.
3. By induction on t it follows that if st $\in$ dom f, then s $\in$ dom f. ◊

Thus we see the precise connection bewteen the types of machine discussed and the types of sequential function.

3.4. Periodic straight move stream X-machines

It is clear that a straight move stream X-machine does not necessarily compute a partial generalised stream function. However, there is a class a straight move stream X-machines which does.

Definition 3.4.1:
A straight move stream X-machine *M* $= (Q, \Sigma, \Gamma, M, \alpha, \beta, \Phi, F, Q_o, T, m_o)$ is called *periodic* if:

> 1. if $p = \varphi_1 \ldots \varphi_n$ is a path with $\varphi_1 \in \{R_\delta\} \times \Phi_M \times (\{L_{-\sigma} | \sigma \in \Sigma \cup \{\delta\}\} \cup \{I\})$, then
> $\varphi_i \in (\{R_\delta\} \cup \{I\}) \times \Phi_M \times (\{L_{-\sigma} | \sigma \in \Sigma \cup \{\delta\}\} \cup \{I\})$ for $i \in \{2, ..n\}$. In other words, the machine cannot produce any output after a blank (δ) has been read except the blank (δ) .
> 2. if $p = \varphi_1 \ldots \varphi_n$ is a loop (i.e. a path from a state q to itself) with $\varphi_i \in \Phi_3 \cup \Phi_4$,
> $i = 1, ..., n$, then $\varphi_i \in \Phi_4$, $i = 1, ..., n$. Therefore, no loop on empty input operations can produce any output.

The periodicity condition does not really affect the computation power of a straight move stream X-machine much since, for example any straight move stream X-machine acceptor (i.e. the output alphabet is $\Gamma = \emptyset$) is periodic. However, it ensures that the function computed is a partial generalised stream function.

Proposition 3.4.2:
Any deterministic periodic straight move stream X-machine computes a partial generalised stream function.

Proof:
It follows by induction. The first condition ensures that the function is segment preserving, the second one that the Lipschitz condition is satisfied. ◊

4. (Straight-move) stream X-machines with stacks

In this section we impose a particular structure on the memory of the stream X-machine and explore the consequences.

4.1. k-stack X-machines
It is fairly clear that the computational power of the X-machine model will depend on the type Φ that the machine works on (one could, for example, use non-computable transition functions, but that is neither useful nor desirable in this context). In what follows we shall introduce and examine several X-machine models whose memory structure is a stack or a finite set of stacks. First, the basic operations on stacks will be the usual 'push' and 'pop'. Then we shall use more complex functions on the memory structure, but throughout we restrict ourselves to using those basic functions that can be computed by very simple X-machines.

Definition 4.1.1:
Let M = (Q, Σ, Γ, M, α, β, Φ, F, Q_o, T, m_o) be an X-machine. If

1. M = $\Omega_1^* \times ... \times \Omega_k^*$ (hence X = $\Gamma^* \times \Omega_1^* \times ... \times \Omega_k^* \times \Sigma^*$, where Ω_1, ... Ω_k are finite alphabets
2. $\Phi = \{\varphi = (\phi_\Gamma, \phi_1,..., \phi_k, \phi_\Sigma,) | \phi_\Gamma : \Gamma^* \to \Gamma^*, \phi_\Sigma : \Sigma^* \to \Sigma^*, \phi_i: \Omega_i^* \to \Omega_i^*$ are partial functions, $\phi \in \Phi_i\}$, where $\Phi_i = \{R_{-u} | u \in \Omega_i\} \cup \{R_u | u \in \Omega_i\} \cup \{I, E\}$, i = 1, ..., k

then M is called a *k-stack X-machine*.
Note: The partial function E: $\Sigma^* \to \Sigma^*$ defined by dom E = {1} and E(1) = 1 checks whether the stack is empty or not.

From the definition above, it is clear that
1. a *k-stack stream X-machine* has the type:
$$\Phi = \{R_\gamma | \gamma \in \Gamma\} \times \Phi_1 \times ... \times \Phi_k \times \{L_{-\sigma} | \sigma \in \Sigma\} \cup \{R_\delta\} \times \Phi_1 \times ... \times \Phi_k \times \{L_{-\delta}\}$$
2. a *k-stack straight move stream X-machine* has the type
$\Phi = \{R_y\} \times \Phi_1 \times ... \times \Phi_k \times \{L_{-x}\}$, where $x \in \Sigma' \cup \{1\}$ and $y \in \Gamma' \cup \{1\}$.

Theorem 4.1.2:
Let Σ and Γ be two alphabets and let f: $\Sigma^* \to \Gamma^*$ be a partial recursive function, then there exists a deterministic 2-stack straight move stream X-machine M which computes f.

Proof:
If f is recursive enumerable, there exists then a Turing machine T with Q = $\{q_1,..., q_n\}$ the state set (q_1 is the Start state), Ω the set of tape symbols ($\Gamma \cup \Sigma \subseteq \Omega$) which computes f. Hence, if t is the initial value of the tape and t' the end one Rmb(t') = f(t), where Rmb: $\Gamma'^* \to \Gamma^*$ is a function which removes all the occurrences of the blank symbol δ from the tape. Any transition of T can be described as (q, a) → (q', a', d), where q is the state T currently is in, a the character read, q' the next state, a' the replacement character and d ∈ {L, R} is the direction the tape head moves in.
We can now simulate the Turing machine T on the following straight move stream X-machine M :
1. The set of states is Q' = $\{q_1', q_1'', ..., q_n', q_n''\} \cup$ Q''. The states set of M is obtained by duplicating each state from Q and adding some extra states. The set Q'' will explicitly follow from the construction of M . M will be in the state q_i', i = 1, ..., n if T is in the state q_i and it has not read a blank (δ) from the tape (therefore the Turing machine has not finished reading the input sequence); M will be in the state q_i'', i = 1, ..., n if T is in the state q_i and it has read a blank (δ) from the tape (the Turing machine has read the whole input sequence).
2. The initial state is q_1' and the set T' = $\{q_i'' | q_i$ is a Halt state of T}
3. The memory is M = $\Omega'^* \times \Omega'^*$, where Ω' = Ω ∪ {δ}. The values of the two stacks s and s' will hold the tape of the Turing machine up to the rightmost location of the tape that has been read by the tape head, i.e. if t = $a_1 ...a_j$, ($a_1,..., a_j \in \Omega'$) , is the tape up to the rightmost location that has been read by the head tape and i is the current position of the tape head, i ≤ j, then s = $a_1 ...a_{i-1}$, s' = $a_j ...a_i$. Hence t = s rev(s'), where rev(x) denotes the reverse of the string x.
The initial value of the memory is m_o = (1, 1).
4. Φ and F results by the following simulation of T on M :
a. For a transition (q, σ) → (p, b, R) in T , σ ∈ Σ, b ∈ Ω' the corresponding transitions

in *M* are:
$F(q', \varphi_1) = p'$, $F(q', \varphi_2) = p'$, $F(q'', \varphi_2) = p''$, where
$\varphi_1 = (I, R_b, E, L_{-\sigma})$, $\varphi_2 = (I, R_b, R_{-\sigma}, I)$.
Therefore, if *T* has not finished reading the input string (*M* is in state q') and s' = 1 then *M* reads a new input character. Otherwise, no input is read and *M* only operates on its stacks.
b. For a transition $(q, a) \rightarrow (p, b, R)$ in *T* , $a \in \Omega - \Sigma$, $b \in \Omega'$ the corresponding transition in *M* is:
$F(q'', \varphi_3) = p''$, where
$\varphi_3 = (I, R_b, R_{-a}, I)$.
Since a is not an input character, *T* has finished reading the input string, therefore *M* operates only on its stacks.
c. For a transition $(q, \delta) \rightarrow (p, b, R)$ in *T* , $b \in \Omega'$ the corresponding transitions in *M* are:
$F(q', \varphi_4) = p''$, $F(q'', \varphi_5) = p''$, where
$\varphi_4 = (I, R_b, E, L_{-\delta})$, $\varphi_5 = (I, R_b, R_{-\delta}, I)$.
Therefore *M* can read the end marker of the input string only if a δ has not been read yet.
d. The transitions $(q, \sigma) \rightarrow (p, b, R)$, $(q, a) \rightarrow (p, b, R)$, $(q, \delta) \rightarrow (p, b, R)$, $\sigma \in \Sigma$, $a \in \Omega - \Sigma$, $b \in \Omega'$ can be obtained from the ones above by replacing φ_i, i = 1, ..., 5 by $\varphi_i' = \varphi_i\ Tf^2$, where Tf is the function which transfers any character from the first stack to the second one. Such transition can be transformed into a sequence of 3 straight move stream X-machine operations by adding two new states $r, r' \in Q''$. For example $F(q', \varphi_1 Tf^2) = p'$ is equivalent to $F(q', \varphi_1) = r$, $F(r, \varphi) = r'$, $F(r', \varphi) = p'$, where $\varphi \in \{(I, R_{-a}, R_a, I) | a \in \Omega'\}$ (i.e. φ takes all the values of the set $\{(I, R_{-a}, R_a, I) | a \in \Omega'\}$).

In order to complete our constructions, we have to deal with the following two problems:
a. *M* has to read the entire input sequence even if *T* halts earlier. This can be easily addressed by adding one extra state $r_i \in Q''$ for each $i \in \{1, ..., n\}$ such that q_i is a Halt state of *T* and the following transitions: $F(q_i', \varphi) = q_i'$, $F(q_i', \varphi') = r_i$, $F(r_i, \varphi'') = r_i$, $F(r_i, \varphi''') = q_i''$, i = 1, ..., n, where $\varphi \in \{(I, R_a, R_{-a}, I) | a \in \Omega'\}$, $\varphi' = (I, I, E, I)$, $\varphi'' \in \{(I, R_\sigma, I, R_{-\sigma}) | \sigma \in \Sigma\}$, $\varphi''' \in \{(I, R_\delta, I, R_{-\delta}) | \sigma \in \Sigma\}$. Therefore if *T* has halted without having finished reading the input string, *M* will store the part of the tape already read into the first stack, read the remaining part (until a δ is reached) and store the remaining part of the tape into the first stack. Since no path can leave a Halt state in *T*, *M* remains deterministic.
b. So far *M* does not produce any outputs. Therefore, any transition of the type $F(q, \varphi) = q_i''$, where $q \in Q'$ (Q' is the state set of *M* constructed so far), $q_i'' \in T'$ (i.e. terminal state) has to be replaced by $F(q, \varphi GH) = q_i''$, where G stores s' rev(s) into s', (therefore s' will hold the reverse of the tape value t) where s and s' are the values of the two stacks, and H outputs Rmb(rev(s')) δ (i.e. the string obtained by erasing all the blanks from the tape t followed by a blank). This can be achieved by adding 3 extra states r_1', ..., $r_3' \in Q''$ for each transition of the type $F(q, \varphi) = q_i''$ and replacing this transition with $F(q, \varphi) = r_1'$, $F(r_1', \varphi_1) = r_1'$, $F(r_1', \varphi_2) = r_2'$, $F(r_2', \varphi_3) = r_2'$, $F(r_2', \varphi_4) = r_2'$, $F(r_2', \varphi_5) = r_3'$, where $\varphi_1 \in \{(I, R_{-a}, R_a, I) | a \in \Omega'\}$, $\varphi_2 = (I, E, I, I)$, $\varphi_3 = (I, I, R_{-\delta}, I)$, $\varphi_4 \in \{(R_\gamma, I, R_{-\gamma}, I) | \gamma \in \Gamma\}$, $\varphi_5 = (R_\delta, I, I, I)$.

From the construction above it is clear that f = $\alpha \circ |\mathcal{M}\ | \circ \beta$. Therefore $\mathcal{M}$ computes f. ◊

Conversely, any simple n-stack straight move stream X-machine can be simulated by a Turing machine. This is achieved by placing the input string, the n stacks and the output string on the Turing machine tape separated by an extra symbol.

Let us denote by F_k and $\mathcal{F}_k$ the class of partial functions computed by k-stack straight move stream X-machines and k-stack generalised stream X-machines respectively and

$F = \bigcup_{k=1}^{\infty} F_k$, $\mathcal{F} = \bigcup_{k=1}^{\infty} \mathcal{F}_k$

Then we have:-

Corollary 4.1.3:
F_n = {f| f is a partial recursive function} ∀ n ≥ 2.

Corollary 4.1.4:
F = {f| f is a partial recursive function}.

Therefore, the hierarchy of the sets F_n stops at 2 and F_2 is the set of all one place partial recursive functions. Obviously, $F_1 \subset F_2$, since a straight move stream X-machine acceptor is equivalent to a push-down automaton.

From proposition 3.1.4, it is clear that a generalised stream X-machine cannot compute an arbitrary partial recursive partial generalised stream function f since the domain of f might not be recursive. Therefore, generally, a 2-stack straight move stream X-machine cannot be converted into a generalised stream X-machine.
However, if the machine is a 1-stack straight move stream X-machine, the conversion could be possible since ∀ f∈ F_1, dom f is a deterministic context-free language. We shall consider the class of periodic 1-stack straight move stream X-machines (since the function computed in this case is necessarily a partial generalised stream function) and we denote by F_1* the restriction of F_1 to the class of periodic straight move stream X-machines.
The first question which arises is whether a periodic 1-stack straight move stream X-machine can be simulated by a k-stack generalised stream X-machine, with k > 1. However, the answer is negative.

Proposition 4.1.5:
$\mathcal{F}_n$ and F_1 * are incomparable ∀ n ∈ N, n ≥ 2.
Proof:
We prove the proposition for X-machine acceptors (i.e. $\Gamma = \emptyset$).
Let Σ = {a, b, c} and L = $\{a^n\ b^n\ c^n \mid n \in N\}$. Since L is not a context free language, L ∉ F_1 *. It can be proved easily that L ∈ $\mathcal{F}_2$.
Conversely, let L' = $\{a^{i1}\ b\ a^{i2}\ b \dots a^{ir-1}\ b\ a^{ir}\ c^s\ a^{ir-s+1} \mid r \geq 1, 1 \leq s \leq r,\ i_j \geq 1$ for all $1 \leq j \leq r\}$. In [2] it is proven that L' ∈ F_1 * and L' ∉ $\mathcal{F}_n$ ∀ n ∈ N. ◊

Hence, the usual push and pop operations are not sufficient for our purpose and therefore we need a more complex basic type Φ. In a future paper, [9], we shall present two new types of generalised stream X-machines with stacks and prove that they compute exactly F_1*. The approach is to consider Φ as a set of partial functions computed by machines which are already well known (for example finite state machines) rather than using simple push and pop operations. This provides us with a new way of building hierarchies of computational models.

5. Conclusions and further work

We have defined a number of stream X-machines, generalised stream X-machines and straight move stream X-machines with stacks and investigated their computation capabilities. Further work might involve generalising some of the models presented in this paper (for example a regular stack X-machine or a stack X-machine with markers with a finite number of stacks) and exploring closure properties of classes of the functions computed.

One of the motivations for pursuing this work is to examine more practical alternatives to the Turing model for system specification. The use of stream X-machines has turned out to be a very powerful and easily used fomalism for describing many different types of software systems and hardware devices. Detailed specifications have been developed for several software systems, graphical user interfaces, real-time systems, robot controller devices and parallel processing chips, [10]. All have the property that it is possible to determine that the specifications represent computable functions and are thus implementable. The theory of testing that we have developed in [6] builds on this fact. Essentially it compares the formal specification, represented as a stream X-machine with the implementation which can also be interpreted as an X-machine. The construction of functional test sets then reduces to the construction of sets that will determine whether two X-machines have the same behaviour. The process of constructing such test sets has now been fully specified and could be automated. The theory developed here, together with other results, has been crucial for the development of a fully general theory and procedure for system testing.

6. References.

[1] S. Eilenberg, (1974), *Automata, languages and machines, Vol. A,* Academic Press.
[2] M. A. Harrison, (1979), *Introduction to Formal Language Theory,* Addison Wesley.
[3] M. Holcombe, (1988), "X-machines as a Basis for System Specification", *Software Engineering Journal,* **3** (2) , 69-76.
[4] M. Stannett, (1990), "X-machines and the halting problem - building a super-Turing machine ", *FACS,* 2, 331-341.
[5] F. Ipate, (1995), PhD Thesis, Computer Science, University of Sheffield.
[6] G. T. Laycock, (1992), "The theory and practice of specification based software testing", Ph.D. Thesis, Department of Computer Science, University of Sheffield.
[7] F. Ipate & M. Holcombe, (1994), "Formal test set generation", submitted.
[8] S. N. Cole, (1971), "Deterministic Pushdown Store Machines and Real-Time Computation", *Journal of the association for computing machinery,* **18** (2), 306-328.
[9] F. Ipate & M. Holcombe, (1995), "X-machines with stacks", Under preparation.
[10] M.Holcombe and coworkers, (1992-5), Various internal reports describing case studies for X-machine specifications, Dept. of Computer Science, University of Sheffield..