Foundations of the X-machine Theory for Testing

Research Report CS-02-06

J. Aguado and A. J. Cowling

Department of Computer Science, Sheffield University
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK

Abstract

The X-machine (also known as Eilenberg X-machine) is an abstract model similar to the finite-state machine (FSM), which contains a data set. The edges between control-states are labelled with relations (functions) that operate on this data. In this way the machine integrates the control structure and the data processing under the same formalism, and moreover, it allows them to be specified separately. The X-machine (EXM) is a general model of computation, in the sense that it can simulate other machine models including the Turing machine. Several classes of EXM have been proposed by means of restrictions on the data set or on the form of the relations (functions).
The stream X-machine (SXM) is a subclass of EXM with capabilities for handling sequences (streams) of inputs and outputs. Even more important is that the Holcombe-Ipate testing approach (SXMT) corroborates the functional equivalence between SXM behaviours (finds all the faults). Some recent investigations have shown that the SXMT can be extended to some other classes of EXM, and to deal with non-determinism. This report presents a comprehensive survey of the theoretical foundations on which the testing approaches for these machines are based.
The applicability of the SXMT to systems of communicating EXM has been studied elsewhere. This report covers two models related to these systems, namely the straight-move stream X-machine (Sm-SXM), and the multiple-stream X-machine (M-SXM). The M-SXM is formalised and analysed here and it is proved that any M-SXM can be reduced to a SXM that computes the same relation.

Keywords: X-machines (EXM), stream X-machines (SXM), generalised stream X-machines(GSXM), straight-move stream X-machines (Sm-SXM), multiple-stream X-machines (M-SXM), formal specification, conditions for testing, testing.

Contents

1. Introduction

Samuel Eilenberg originally introduced the X-machine model in 1974 in his study of automata theory [1] as a more tractable alternative to finite-state machine (FSM), finite automata (FA), pushdown automata (PDA), Turing-machine (TM) and other kind of machine models. In fact, Eilenberg demonstrated that his model is general enough to embrace these abstract machines. It was not until 1988, however, that the formalism was used as a possible hierarchical refinement specification language by Holcombe [2], who later took a step forward by providing the basis for an approach of refinement that integrates both specification and testing under one paradigm; the X-machine [3]. Since then, much work has been done and applied to a large number of specification and modelling problems (*e.g.* [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]). Nevertheless, our main interest here is more concerned with the testing aspects derived from the X-machine theory, rather than its related specification techniques. Note that these mechanisms have been referred to in the literature as X-machines, as they were originally named. It has been suggested that (reported in [21]) their name to be changed to *Eilenberg machines*. Therefore, the name (Eilenberg) X-machines will be used throughout this report.

The concept of an Eilenberg X-machine (EXM) is straightforward and is based on the concept of a FSM, which controls actions usually referred to as *processing relations or functions* on a data set. In this way, the model integrates the control and the data processing, while at the same time allowing them to be specified separately [22] and therefore the data space is independent of the control structure [23]. In this report, we will simply refer to both types of actions (*i.e.* processing relations and functions) as functions, making the distinction between them when necessary. The presence of (partial) functions and a data set introduces the possibility of defining a number of classes of EXM. This feature has resulted in different generalisations of the concept, which have been identified and analysed in [8, 9, 22, 23, 24, 25]. These types of machines have been fundamentally defined by means of restrictions on the data set, and on the form of the functions.

One of these classes is the (Eilenberg) stream X-machine (SXM), which according to [26] is close to some of Eilenberg's initial thoughts. This subclass of EXM was not explicitly proposed until 1993 [23, 24]. In a SXM, the input and the output are streams of symbols and the machine has a memory that takes values from a particular (may be infinite) set. The SXM processes its input stream in an orderly manner, which determines the next control-state, depending on the current control-state and the memory value. At the same time, the machine produces an orderly stream of outputs and updates its internal memory.

The importance of the SXM is derived from the fact that the generality of the EXM may be an obstacle in developing a testing method based on it. However, the SXM is general enough to cope with a wide range of computational problems as long as it is restrictive enough to provide a basis for a testing theory [25]. In fact, a reliable technique for testing SXM has been developed [22, 25, 27]. This method, the Holcombe-Ipate testing approach (SXMT), in its simplest form involves identifying test cases from the different paths that a SXM (specification) can take through its set of control-states, and then testing the response of another SXM (implementation) to all these sequences generated from the specification. More specifically, it is assumed that from the implementation under test (IUT) it is only possibly to observe the outputs generated by it, on receipt of inputs (*i.e.* there is no information about the internal structure of the IUT). Because of that, the SXMT can be considered a black-box testing technique. Naturally, some extensions of the SXMT have been proposed to deal with more generalised classes of SXM [28, 29, 30], and with non-determinism [31, 32, 33, 34, 35, 36]. This report presents a comprehensive survey of the foundations that support these testing approaches developed from EXM.

In addition, recent investigations have shown that this SXMT can be extended to some of the communicating EXM. In particular, there is a model called *communicating (Eilenberg) stream X-machine systems* (CSXMS) [37], which has established the basis for further developments in providing communication capabilities for the EXM and testing systems specified in this way (*e.g.* [38, 39, 40, 41, 42, 43, 44, 45, 46, 47]). From the testing perspective, one of the most important results is that it is possible to construct a stand-alone EXM from any CSXMS with the same input-output relationship [37]. This in principle allows the application of SXMT to CSXMS where the testing process from a CSXMS will consist of constructing an equivalent EXM and then applying the testing to it. However, this transformation could result in a machine that supports *empty-operations* [46, 47, 48] and the class of EXM that allows these operations is the *straight move (Eilenberg) stream X-machine* (Sm-SXM).

Another approach for communication called the *modular specification of systems using (Eilenberg) communicating X-machines* (MSS) proposed in [44] was originally based on some syntactical modifications of the *X-machine description language* (XMDL) [49, 50], a descriptive language for EXM. In [46] we justify a different practice for specifying systems based on MSS, in order to provide reference information within the same or at least a similar standard of mathematical notation to that that has been used in the body of the EXM theory. With regard to this, the *(Eilenberg) multiple-stream X-machine* model (MSXM) is proposed here (as far as we know for the first time) and the equivalence between MSXM and SXM is demonstrated, implying that the SXMT can be applied without modification to the MSXM. Thus, this report would not completely accomplish its objective, if the Sm-SXM and MSXM models were not included.

The rest of the report is organised as follows: Section 2 introduces the EXM, and in particular section 2.1 gives its basic concepts. In section 2.2, we study simple but useful properties that deterministic machines have, and the relationship of determinism with the associated FA of an EXM. Subsequently, in section 2.3 the notion of trivial path for an EXM with the data set having the same form as the SXM data set is briefly discussed, and more importantly, it is explained why trivial paths can be a source of non-determinism. Next, in section 2.4 the notion of completely specified EXM is presented. Because an EXM has a set of functions and a set of data, a machine could be completely specified with respect to any or both of these sets. In section 2.5, we discuss some methods for constructing EXM that can simulate FA, PDA or TM.

Section 3 introduces the concepts and properties of the SXM and, as in the rest of the report; the presentation is strongly oriented to testing. In section 3.1, the model is reviewed indicating the differences found in the literature. While there are some minor discrepancies, only two of them deserve special attention. Firstly, in some cases the order in which the concatenation of the output symbol is carried out with respect to the stream can differ. Secondly, the termination condition that is commonly accepted establishes that the input stream must be empty; nevertheless, this was not always the case, especially in the earlier references. Thus, in order to facilitate this analysis, we use a new approach for the streams, which consists of defining them as abstract data types (ADT).

In section 3.2 the concepts of the relation computed by an SXM, and the (extended) the transition relation of a SMX that respectively characterise the outputs corresponding to an input, and all the transitions of a machine, are introduced. Additionally, in the same section the notions of configuration and change of configuration are reviewed, the former includes information about the elements of the data set, and the control-state, the latter describes under which conditions a change of configuration can occur. Furthermore, we prove the equivalence between such concepts. Section 3.3 presents a general introduction to the SXMT indicating the type of faults that the method finds, and the five conditions that the specification machine and the IUT machine have to fulfil are described. Of these, the first two conditions are discussed in the same section. These are that both machines have the same set of processing functions and that if there is a different number of states between them, this number is known.

The other three conditions are studied in the following sections. Specifically, section 3.4 deals with determinism, section 3.5 has to do with reachability of states and attainability of memory values, and section 3.6 presents in detail the properties known as *design-for-test* conditions. Often when a test is performed from the observation of the output sequence produced by the machine, a number of particularities may need to be determined. For instance, given a SXM we may want to know in which state it is, which processing function has been used, or what is the memory value. So an input sequence can be applied to the machine and from its input/output behaviour, information about this can be deduced. The properties of SXM reviewed in this section are *completeness* and *output-distinguishability*. Completeness guarantees that any path of the machine can be followed from an initial state and an initial memory value. Output-distinguishability establishes that it is always possible to distinguish between any two distinct transitions that have been applied to the machine. These two conditions, however, can be relaxed to the subset of memory that can be reached from an initial state without invalidating the SXMT [47]. When this is done, they are called *relaxed-variants* of the design-for-test conditions, which are also included in section 3.6. Section 3.7 is concerned with the last assumption of the SXMT namely, the minimality of the associated FA of an SXM.

Section 4 considers other classes of EXM related to testing. The GSXM is introduced in section 4.1, because a generalisation of the SXMT requires different design-for-test conditions, these are studied here. In addition, in section 4.1 the notions related to testing non-deterministic GSXM (NGSXM) are presented. In section 4.2, the M-SXM is presented and it is proved that the model has the same computational power as the SXM model, which in principle allows the application and extension of existing testing strategies for SXM. Section 5 summarises the conclusion of the report.

2. (Eilenberg) X-machines (EXM)

An EXM has a set of control-states 'Q' (*i.e.* like a FSM) and a basic data set 'X'. Usually, this data set has the form of a triplet $X = \Gamma^* \times M \times \Sigma^*$ where Γ^* and Σ^* are sequences of symbols from an output and an input alphabet and M is a possible infinite set called the memory of the machine. The transitions between states contain processing functions from a set 'Φ' (called the *type* of the machine) that operate on this data (*i.e.* X). Thus, the type of the machine defines the elementary operations that the machine is capable of executing on the data. A computation of an EXM can be seen as a traversal of a path in the space-state, together with the application of the corresponding processing functions (which label the edges of the path) to successive values of the fundamental data set starting from a specified initial value.

2.1 The EXM model

Formally, the most general definition of an EXM is as follows [2, 22, 23, 24, 25, 26, 51]:

Definition 1: An EXM is a 10-tuple $\Lambda = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T)$ where:

- X is the fundamental data set over which the machine operates.
- Y and Z are respectively the external input and output sets.
- α and β are respectively the input and output coding relations where:
  $\alpha: Y \leftrightarrow X$
  $\beta: X \leftrightarrow Z$
- Q is the (finite) set of states.
- Φ is the type of Λ a set of non empty relations on X:
  $\Phi: P\,(X \leftrightarrow X)$
- F is the next-state (partial) function, which is often described by means of a state-transition diagram:
  $F: Q \times \Phi \rightarrow P(Q)$
- I and T are respectively the sets of initial and final states:
  $I \subseteq Q, T \subseteq Q$

The input and output relations are used to codify the input and the output sets into, and from, the fundamental data set X, therefore they provide a general interface mechanism. Since one function $\phi \in \Phi$ is associated with each transition of the machine only a finite subset $\Phi' \subseteq \Phi$ will be used in a particular EXM because it has only a finite number of transitions. The EXM constitutes a widely accepted and well-established model and there are a number of different types of this model. Nevertheless, definition 1 and its associated concepts can be regarded as the starting point from which all the other EXM models emerge. The operation of an EXM can be described as follows:

1. The machine reads an input $y \in Y$, and the input relation codifies it as an element of the data set, that is to say $\alpha(y) \rightarrow x_0$ such that $x_0 \in X$ is an initial data-state.
2. From an initial control-state $q_0 \in I$, the machine selects an edge that goes from q_0 to $q_1 \in Q$ labelled $\phi_1 \in \Phi$ such that ϕ_1 can act over x_0. The machine moves to the new control-state q_1 and computes $\phi_1(x_0)$.
3. This process continues in the same way from state q_1, that is the machine chooses an edge, labelled $\phi_2 \in \Phi$, which goes from q_1 to $q_2 \in Q$ such that $\phi_2(\phi_1(x_0))$ can be determined and the machine moves to q_2.
4. When no further computation is possible and the machine has reached a final state $q_f \in T$ the final value $x_f \in X$ is decoded as an element of the external output set, that is $\beta(x_f) \rightarrow z$ where $z \in Z$.

Clearly, an EXM accepts only a single input and provides just a single output at the end.

The associated FA of an EXM can be defined as A = (Φ, Q, F, I, T) where Φ is viewed as an abstract alphabet. Consequently, it is natural that many concepts of FSM also apply to the EXM model. One of them is the concept of a *path*, which is related to the same notion in automata and graph theory and forms the basis for the definitions of the *behaviour of* an EXM and the *relation computed by* an EXM.

Definition 2: If $q, q' \in Q$, $\phi \in \Phi$ and $q' \in F(q, \phi)$, it is said that ϕ is an edge, arc or arrow from q to q' and it is denoted by $\phi: q \rightarrow q'$. A path p from $q \in Q$ to $q' \in Q$ is an alternating sequence of edges and states, such that every edge goes from the state before it to the state after it: $p = <q, \phi_1, q_1, \phi_2, q_2, \phi_3, q_3, \dots, q_n, \phi_{n+1}, q'>$. In [23, 24, 51] a path is represented as $p = (<q, q_1, q_2, \dots, q_n, q'>, <\phi_1, \phi_2, \dots, \phi_{n+1}>)$ and

> The component $<q, q_1, \dots, q_n, q'>$ is the *sequence of states* of the path or the *Q-path* of p, denoted by p_Q,
> The component $<\phi_1, \phi_2, \dots, \phi_{n+1}>$ is the *sequence of edges* or the *Φ-path* of p denoted by p_Φ.

The main reason for representing a path with these two components is that distinct sequences of states might have the same sequence of edges and *vice versa*. That is, for the same *Φ-path* there could be more than one *Q-path* associated with it. However, when the state sequence is irrelevant, it can be assumed that the path is given by the *Φ-path*. A path is a *successful* or *full* path if and only if it starts in an initial state $q \in I$ and ends in a final state $q' \in T$, otherwise the path is said to be *partial*. A *loop* is a path that starts and ends in the same control-state.

Definition 3: For any path (*Φ-path*) $p = <\phi_1, \phi_2, \dots, \phi_{n+1}>$ the composite (partial) function computed by the EXM when it follows that path is $|p| = \phi_{n+1} \bullet \phi_n, \dots, \phi_2 \bullet \phi_1 \in X \leftrightarrow X$ where $|p|$ is also called the *label* of p. The identity function of the label of any path corresponds to an empty sequence of processing functions.

Convention 1: The symbol δ indicates an empty sequences of processing functions, in this form $\forall\, x \in X$, $|\delta|(x) = x$.

The union of all the computations carried out over successful paths gives the behaviour of an EXM or more formally [22]:

Definition 4: The behaviour of an EXM Λ is the relation $|\Lambda|: X \leftrightarrow X$ defined as $x\, |\Lambda|\, x' \Leftrightarrow \exists\, q \in I$, $q' \in T$ and a path $p: q \rightarrow q'$ such that $|p|(x) = x'$.

The behaviour can be seen alternatively as the relation $|\Lambda|: X \leftrightarrow X$, defined as $|\Lambda| = \cup\, |p|$ with the union extending over all successful paths in Λ, from [25]:

Definition 5: The relation computed by an EXM is given by $f = \beta \bullet |\Lambda| \bullet \alpha: Y \leftrightarrow Z$.

Example 1: Let us consider the problem of finding the scalar product of two positive integer vectors of the same order. The formulation is as follows: Given $v_1 = <a_1, a_2, \dots, a_n>$ and $v_2 = <b_1, b_2, \dots, b_n>$ where $\forall\ 1 \le i \le n$, $a_i, b_i \in \mathbf{N}^+$ the scalar product is $\pi = v_1 \bullet v_2 = \sum_{i=1}^{n} (a_i \text{ x } b_i)$.

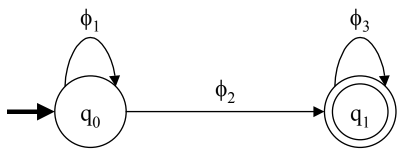
In order to model this with an EXM Λ = (X, Y, Z, α, β, Q, Φ, F, I, T) the fundamental data set and the input and output data types are given by $X = \mathbf{N} \text{ x } \mathbf{N} \text{ x } \mathbf{N}^+ \text{ x } A$, $Y = \mathbf{N} \text{ x } A$ and $Z = \mathbf{N}$. The fundamental data set X is a 4-tuple where the first component will be used as a counter, due to which it must be a natural number. The second element keeps the result of the computation so it is also a natural number. The third component contains the size n of the vectors thus it is a positive integer (*i.e.* $n \ge 1$). The fourth component $A = \mathbf{N}^+ \text{ x } \mathbf{N}^+ \text{ x } \dots$ is a possible infinite sequence of positive integers (*i.e.* Cartesian product). The reason for defining A in this way is to ensure that any arbitrary pair of vectors of order n will fit into the type (*i.e.* to avoid the out of memory error). The elements of the vectors will be arranged in the following order $a_1, b_1, a_2, b_2, \dots, a_n, b_n$.

The external input set Y is defined as a pair. The first element is a natural number that corresponds to the order of vectors and the second component is as in the data set X (*i.e.* the set A). The external data set Z is equal to the set of natural numbers because it will contain the result (scalar product) calculated by the machine. In what follows we will use the projection functions π_1, π_2, π_3 and so on in the usual manner. That is, for all $1 \le i \le n$, $\pi_i: S_1 \text{ x } S_2 \text{ x } \dots \text{ x } S_n \rightarrow S_i$ where all S_i are sets. The input and output coding relations can be defined by:

$\alpha(n, a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n) = (1, 0, n, (a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n,))$ and

$$\beta(c, r, n, a_1, a_2,\ldots, a_n, b_1, b_2,\ldots, b_n) = \pi_2(c, r, n, (a_1, a_2,\ldots, a_n, b_1, b_2,\ldots, b_n)) = r$$

The rest of the specification is presented in figure 1. The machine has two loops and one function, the first one ϕ_1 computes the multiplication of every pair a_i x b_i, then function ϕ_2 resets the counter used for controlling the loops and finally the second loop ϕ_3 calculates the summation. Additionally, from definition 2 it is easy to see that ϕ_2: $q_0 \rightarrow q_1$ is an edge and $p = (<q_0, q_0, q_1, q_1>, <\phi_1, \phi_2, \phi_3>)$ is a path where its Q-path is given by $p_Q = <q_0, q_0, q_1, q_1>$ and its Φ-path is $p_\Phi = <\phi_1, \phi_2, \phi_3>$.



$\Phi = \{\phi_1, \phi_2, \phi_3\}$

$\phi_1(x) = x'$, if $\pi_1(x) \leq \pi_3(x)$ where:
    $\pi_1(x') = \pi_1(x) + 1$
    $\pi_i(x') = \pi_i(x) * \pi_{i+1}(x)$ with $i = \pi_1(x) * 2 + 2$

$\phi_2(x) = x'$, if $\pi_1(x) > \pi_3(x)$ where:
    $\pi_1(x') = 1$

$\phi_3(x) = x'$, if $\pi_1(x) \leq \pi_3(x)$ where:
    $\pi_1(x') = \pi_1(x) + 1$
    $\pi_2(x') = \pi_2(x) + \pi_i(x)$ with $i = \pi_1(x) * 2 + 2$

$Q = \{q_0, q_1\}$
$I = \{q_0\}$ and $T = \{q_1\}$

$F = \{f_1, f_2, f_3\}$ where:
    $f_1(q_0, \phi_1) = q_0$
    $f_2(q_0, \phi_2) = q_1$
    $f_3(q_1, \phi_3) = q_1$

Figure 1

It must be noted that the specification of the EXM could be done using only a one-state transition diagram with only one function $\phi \in \Phi$ that performs the additions and multiplication required by the computation. However, the objective here is to provide a specification that allows the illustration of as many aspects as possible. The machine has two states where q_0 and q_1 are respectively the initial and final states. In the transition diagram the initial state is indicated by an incoming arrow as long as the final state is represented by a double circle, as it is done in the notation in common use in automata theory.

The machine Λ starts its operation in state q_0 since it is the only initial state. In this state, two processing functions emerge from it, namely ϕ_1 and ϕ_2. Nevertheless, because *domain*(ϕ_1) = $\{x \in X \mid \pi_1(x) \leq \pi_3(x)\}$ and *domain*(ϕ_2) = $\{ x \in X \mid \pi_1(x) > \pi_3(x)\}$ then *domain*(ϕ_1) $\cap$ *domain*(ϕ_2) = $\varnothing$ implying that one but not the other can be applied at a given time. The projection function $\pi_1(x)$ provides the value of a natural number that is employed as a counter and that is initially 1, and the projection function $\pi_1(x)$ returns the order of the vectors. Therefore, the function ϕ_1 will be applied again and again while the value of the counter is less than or equal to the order of the vectors and once this condition is not longer hold the function ϕ_2 will be computed moving the machine to the control state q_1.

Observe that function ϕ_1 increments the counter by one, while at the same time it computes the multiplication of the corresponding elements of the vectors. That is, for a particular value of the counter $i = \pi_1(x)$, $a_i \leftarrow a_i * b_i$ because $\pi_i(x) = a_i$ and $\pi_{i+1}(x) = b_i$. The function ϕ_2 simply reinitialises the counter value (*i.e.* $\pi_1(x)$) to 1 in order to control a new loop for calculating the summation and that takes place in state q_1. The processing function ϕ_3 has the same domain as ϕ_1, that is *domain*(ϕ_3) = $\{x \in X \mid \pi_1(x) \leq \pi_3(x)\}$ or in other words this function will be executed until the value of the counter is bigger than the order of the vectors. The function increases the counter $i = \pi_1(x)$ and computes $r \leftarrow r + (a_i * b_i)$ where $\pi_2(x) = r$. Initially $\pi_2(x) = 0$, and $\pi_i(x) = a_i * b_i$ after the computation of the first loop.

Finally, the machine returns the value of $\pi_2(x)$ (*i.e.* the result) using for that the decoding function β. As an illustration of how the machine operates consider the problem instance $y = (3, 1, 5, 3, 4, 2, 7) \in Y$ which corresponds to v_1 = <1, 3, 2> and v_2 = <5, 4, 7>. At the beginning, the instance y is trivially coding up to a data set x_0 by means of α as α(3, 1, 5, 3, 4, 2, 7) = (**1**, **0**, 3, (1, 5, 3, 4, 2, 7)). The rest of the process carried out by Λ is presented in table 1.

State	Data set value	Transition function	Processing function
q_0	x_0 = (1, 0, 3, 1, 5, 3, 4, 2, 7)	f_1	ϕ_1
q_0	x_1 = (**2**, 0, 3, **5**, 5, 3, 4, 2, 7)	f_1	ϕ_1
q_0	x_2 = (**3**, 0, 3, 5, 5, **12**, 4, 2, 7)	f_1	ϕ_1
q_0	x_3 = (**4**, 0, 3, 5, 5, 12, 4, **14**, 7)	f_2	ϕ_2
q_1	x_4 = (**1**, 0, 3, 5, 5, 12, 4, 14, 7)	f_3	ϕ_3
q_1	x_5 = (**2**, **5**, 3, 5, 5, 12, 4, 14, 7)	f_3	ϕ_3
q_1	x_6 = (**3**, **17**, 3, 5, 5, 12, 4, 14, 7)	f_3	ϕ_3
q_1	x_7 = (**4**, **31**, 3, 5, 5, 12, 4, 14, 7)	f_3	ϕ_3

Table 1

Once Λ is in final control-state q_2 with a data set value x_7 = (4, 31, 3, 5, 5, 12, 4, 14, 7) where no processing is longer available, the decoding function is used in order to obtain β(4, 31, 3, 5, 5, 12, 4, 14, 7) = 31. Therefore, it is clear that the paths that the machine follows are $p_1 = <\phi_1, \phi_2, \phi_3>$, $p_2 = <\phi_1, \phi_1, \phi_2, \phi_3, \phi_3>$, $p_3 = <\phi_1, \phi_1, \phi_1, \phi_2, \phi_3, \phi_3, \phi_3>$ and so on. Thus, the composite (partial) function computed by Λ when it receives two vectors of order n is as follows. Let $p^1 = <\phi_1, \phi_1, \ldots, \phi_1, \phi_1>$ be the part of the path that corresponds to the application of ϕ_1 n times and let $p^3 = <\phi_3, \phi_3, \ldots, \phi_3, \phi_3>$ be similarly the part of the path that corresponds to the repeated application of ϕ_3 n times, then $|p| = |p^1| \bullet \phi_n \bullet |p^3|$. Clearly, x_0 = (1, 0, 3, 1, 5, 3, 4, 2, 7) $|\Lambda|$ x_7 = (4, 31, 3, 5, 5, 12, 4, 14, 7) is in the behaviour of the machine and f(3, 1, 5, 3, 4, 2, 7) = 31 is in the function computed by Λ. In general, given two vectors of order n, $v_1 = <a_1, a_2, \ldots, a_n>$ and $v_2 = <b_1, b_2, \ldots, b_n>$ this function f: Y $\leftrightarrow$ Z where Y = **N** x A and Z = **N** calculates: $f(n, a_1, b_1, a_2, b_2, \ldots, a_n, b_n) = v_1 \bullet v_2 = \sum_{i=1}^{n} (a_i \times b_i)$.

■

2.2 Deterministic EXM

The sequences of edges in a machine determine the processing of the data set and therefore the function (relation) computed by the EXM. Hence, the complete operation of an EXM can be represented by $\beta(|p|(\alpha(y)))$ where $y \in$ Y. The concept of *determinism* has been defined for the EXM in order to characterise those machines for which $f = \beta \bullet |\Lambda| \bullet \alpha$ is a (partial) function and consequently for any $y \in domain(f)$ only one output will be produced. Clearly, for an EXM to be deterministic the first two necessary conditions that have to be fulfilled are:

1. There will be no more than one initial state and
2. If there are two paths p and p' starting from state q such that they have identical Φ-paths then $p = p'$.

Put simply, these conditions refer to the determinism of the associated FA of the EXM.

Definition 6: The associated FA of an EXM is deterministic (DFA) if:

- There is just one initial state: I = $\{q_0\}$
- F maps each pair (q, ϕ) into no more than one state; F: Q x $\Phi \rightarrow$ Q

The idea of determinism has both a weak and a strong form, which were respectively called *determinism* and *full-determinism* in [24]. Because the weak form has not been longer used, the term that has been commonly accepted for the strong form is determinism. In order to avoid confusion, we shall present these two concepts separately and briefly discuss their difference.

Definition 7: An EXM Λ is deterministic if:

- The associated FA of Λ is a DFA
- α and β are functions:
  α: Y $\rightarrow$ X

β: X $\rightarrow$ Z

- Φ contains only (partial) functions operating on X:
  Φ: *P* (X $\rightarrow$ X)

The conditions of definition 7 do not (on their own) ensure that *f* is a (partial) function. To illustrate this, consider any DEXM. It could be the case that in a particular state and for a value $x \in$ X two different functions ϕ and ϕ' can be applied to it, and thus the selection of one or the other would be made in an arbitrary manner. To avoid this kind of situation an additional restriction with respect to the domain of the functions is required, this is formally determined by the following:

Definition 8: An EXM Λ is *full-deterministic*, if Λ is deterministic and the intersection of the domains of all the functions emerging from a given state, is empty. That is, $\forall\ q \in$ Q, ϕ, $\phi' \in \Phi$, if (q, ϕ), $(q, \phi') \in$ F then $domain(\phi) \cap domain(\phi') = \varnothing$

In words, in a full-deterministic EXM there is at most one possible transition to follow, since for each value of the data set and each control-state, the associated transition (edge) has disjoint domains for its function. Conversely, in a non-deterministic EXM (NDEXM) there may be more than one possible transition to follow. As has been mentioned, the terminology in common use does not make any distinction between determinism and full-determinism, therefore in what follows we will use the word determinism to the concept of definition 8 and we denoted a deterministic EXM as DEXM.

Example 2: The EXM of figure 1 is a DEXM, since from figure 1 it is possible to see that:

1. Its associated FA is a DFA,
2. α and β are functions,
3. the type of the machine contains only functions and
4. $domain(\phi_1) \cap domain(\phi_2) = \varnothing$.

■

2.3 Trivial paths and determinism

In general EXM with a type that has the form X = Z x M x Y are considered here. Note that the triplet is composed as *output x memory x input* (not as *input* x *memory* x *output*) because this is the form that is frequently employed in the literature, also sometimes the input and the output are composed by sequences of symbols of some alphabets Σ and Γ respectively. If this is the case, the data set can be expressed as X = Γ^* x M x Σ^* and therefore, relations of the form *f*: $\Sigma^* \leftrightarrow \Gamma^*$ will be computed by the machine. From this, three projection functions can be obtained in a direct manner.

Definition 9: For X = Γ^* x M x Σ^* we have;

the output projection π_{out} : Γ^* x M x $\Sigma^* \rightarrow \Gamma^*$ (also called *out* or π_1 in the literature),
the memory projection π_{mem} : Γ^* x M x $\Sigma^* \rightarrow$ M (also called *mem* or π_2 in the literature) and
the input projection π_{in} : Γ^* x M x $\Sigma^* \rightarrow \Sigma^*$ (also called *in* or π_3 in the literature)

with the form:

$\forall\ g^* \in \Gamma^*, m \in$ M, $s^* \in \Sigma^*$,
$\pi_{\text{out}}(g^*, m, s^*) = g^*$
$\pi_{\text{mem}}(g^*, m, s^*) = m$
$\pi_{\text{in}}(g^*, m, s^*) = s^*$

Now, since the type of the machine has the form Φ: *P* (Γ^* x M x $\Sigma^* \rightarrow \Gamma^*$ x M x Σ^*) then any function $\phi \in \Phi$ can affect any elements of Γ^* or Σ^*, in any order. From this, a path *p* in an EXM is called a *trivial path*, if for some $x = (g^*, m, s^*) \in$ X when the machine follows that path, $s^* \in \Sigma^*$ is not changed while possibly $g^* \in \Gamma^*$ and $m \in$ M are changed (but not necessarily).

Definition 10: Let *p* be a path of an EXM, if $\exists\ x \in$ X such that $x \in domain(|p|)$ and $\pi_{\text{in}}(|p|(x)) = \pi_{\text{in}}(x)$ then *p* is a *trivial path*.

For any EXM with type X = Γ* x M x Σ* we have the following general result [1] (reported in [25]). Note that this demonstration is ours.

Proposition 1: If Λ is a DEXM and no trivial path connects two terminal states then Λ computes a (partial) function.

Proof. Let us assume that the proposition is true. Now, let $x_0 = (g^*, m, s^*) \in X$ be the initial data set value from which the machine starts the computation. From definitions 7 and 8, the EXM can traverse one and only one path while computing x_0. Let *q* be the state where the computation cannot proceed further, and let $x = (g'^*, m', s'^*) \in X$ be the last value calculated for the data set. Since the machine computes relations of the form $f: \Sigma^* \leftrightarrow \Gamma^*$, it follows that if $q \in T$ then $f(s^*) = g'^*$ and if $q \notin T$ then $f(s^*) = \perp$ implying that Λ computes a partial function.

If for the same machine we include a trivial path *p* from $q \in T$ to another terminal state $q' \in T$ such that the final data set value $x = (g'^*, m', s'^*) \in domain(|p|)$ with $|p|(x) = (g''^*, m'', s'^*)$ this implies that $f(s^*) = \{g'^*, g''^*\}$ is a relation.

□

In conclusion, an EXM Λ with type of the form X = Γ* x M x Σ* computes a (partial) function if Λ is a DEXM and no trivial paths connects terminal states.

2.4 Completely specified EXM

The correctness of the test set generated by the SXMT depends on certain constraints imposed on the design of a machine (see [22, 25, 27]), where one of them corresponds to the notion of *completeness*. Although the SXMT was developed for a subclass of EXM, this concept of completeness was first used for EXM in [24]. Nevertheless, the definition of completeness employed for testing purposes (see definition 31), refers to a different property from the one described in [24], which is more related to the *completeness of the specification* presented in [34] (see definition 32). Thus, to avoid any confusion we change the word "complete" for the word "completely-specified" presented in the original source, and we shall say that:

Definition 11: An EXM Λ is completely-specified with respect to X, denoted by $\Lambda \uparrow X$, if $\forall\, x \in X$ and $\forall\, q \in Q$, $\exists\, \phi \in \Phi$ such that $x \in domain(\phi)$ and $(q, \phi) \in domain(F)$. An EXM Λ is completely-specified with respect to Φ, denoted by $\Lambda \uparrow \Phi$, if $\forall\, \phi \in \Phi$ and $\forall\, q \in Q$ we have that $(q, \phi) \in domain(F)$.

$\Lambda \uparrow X$ ensures that there is always a function, which can be applied to any control-state and any value of the data set. Moreover, $\Lambda \uparrow \Phi$ establishes that all the functions of the type emerge from each one of the control-states. In automata theory, if the transition function is *fully-defined* this means that for each state there is at least one edge for each symbol in the alphabet [52], which of course is closely related to the idea of complete specification. If X is viewed as an abstract alphabet then $\Lambda \uparrow X$ guarantees that for each state there is at least one edge for every possible value of X. On the other hand, let A = (Φ, Q, F, I, T) be the associated FA of Λ thus if F is fully-defined this implies that $\Lambda \uparrow \Phi$. Using these concepts, Laycock [24] proved the following:

Proposition 2: If Λ is a DEXM and $\Lambda \uparrow X$, then for each $q \in Q$ the domains of the functions emerging from it form a (disjoint) partition of X.

Proof. Let $q \in Q$ be any state of Λ where $(q, \phi_1), (q, \phi_2), \ldots, (q, \phi_n) \in F$, by definition 8, it follows that $domain(\phi_1) \cap domain(\phi_2) \cap \ldots \cap domain(\phi_n) = \varnothing$ and by definition 11 we immediately deduce that $domain(\phi_1) \cup domain(\phi_2) \cup \ldots \cup domain(\phi_n) = X$.

□

To go further, we combine these two properties as follows:

Definition 12: An EXM Λ is *complete-specified* denoted by $\Lambda \uparrow (X, \Phi)$ if and only if $\Lambda \uparrow X$ and $\Lambda \uparrow \Phi$.

From this, it is not difficult to see that;

Proposition 3: If Λ is an EXM with $\Lambda \uparrow \Phi$ but **not** $\Lambda \uparrow X$ then there is a subset $X' \subset X$ such that $\Lambda \uparrow (X', \Phi)$.

Proof. Because Λ does not hold the property $\Lambda\uparrow X$ this implies that there is at least one value $x \in X$ such that for any state q and any memory value m there is no function ϕ emerging from q with $(m_0, x) \in domain(\phi)$. Moreover, since $\Lambda\uparrow\Phi$ all the functions of Φ emerge from every state, therefore $x \notin domain(f)$ where f is the relation computed by Λ. Let X" be a subset of X such that $\forall x'' \in X, x'' \notin domain(f)$ and let us modify the machine Λ by redefining the data set as X' = X/X". Clearly this modification does not affect the property $\Lambda\uparrow\Phi$, so for every state

$q, (q, \phi_1) \in F, (q, \phi_2) \in F, \dots, (q, \phi_n) \in F \ \forall\ 1 \leq i \leq n, \phi_i \in \Phi$ with
$domain(\phi_1) \cup domain(\phi_2) \cup \dots \cup domain(\phi_n) = X'$, and

thus $\Lambda\uparrow X'$ and the domain of the relation computed by the machine is X'.

□

Example 3: Clearly, the EXM Λ presented above in figure1, is neither $\Lambda\uparrow X$ nor $\Lambda\uparrow\Phi$. For the first case, we have that for some $x \in X$ where $\pi_1(x) > \pi_3(x)$ in state q_1 there is no $\phi \in \Phi$ emerging from that state such that ϕ can be applied to x. For the second case, there is no edge labelled ϕ_3 that emerges from q_0, and there are no edges labelled ϕ_1 and ϕ_2 emerging from q_2. This is in part a result of the use of two-state machine. However, it is possible to modify the specification for it to be a one-state machine where $q \in Q$, $q \in I$ and $q \in T$ with only one processing function $\Phi = \{\phi\}$ such that $(q, \phi) \in F$ and

$\forall x \in X, \phi(x) = x'$, if $\pi_1(x) \leq \pi_3(x)$ where:
$\pi_1(x') = \pi_1(x) + 1$ and
$\pi_2(x') = \pi_2(x) + (\pi_i(x) * \pi_{i+1}(x))$ with $i = \pi_1(x) * 2 + 2$

Obviously, for this modified version of the machine Λ, we have that $\Lambda\uparrow\Phi$, and Λ is still a DEXM. However, since there are some values of the data set for which there are no a functions from Φ that can be applied to them then the machine does not fulfil the property $\Lambda\uparrow X$. So following the ideas of our proposition 3 we can identify the set X" = $\{ x \mid x \in X$ and $\pi_1(x) > \pi_3(x)\}$ and by defining X' = X/X", we obtain a machine Λ' identical to Λ but with X' as its fundamental data set and therefore:

1. Λ' is a DEXM,
2. $\Lambda'\uparrow(X', \Phi)$ and
3. Λ' computes $v_1 \bullet v_2$.

■

2.5 The EXM and other computational models

Eilenberg [1] (reported in [22]) demonstrated that the EXM is a general model of computation, since FA, PDA, and TM are all special cases of it. Let us present these results but with different proofs from the ones studied in [22]. The reason for doing this is not only to show the relationship between the EXM and the other computational models, but also to review the definitions of the concepts corresponding to these other machine models.

Lemma 1: For each FA there exists an EXM that accepts the same language.

Proof. Let A = $(\Sigma_A, Q_A, F_A, q_A, T_A)$ be a FA where:

- Σ_A is an nonempty, finite set of symbols called the machine's alphabet.
- Q_A is a finite set of states.
- F_A is the transition (partial) function, $F_A : Q_A \times \Sigma_A \rightarrow P(Q_A)$.
- $q_A \in Q_A$ is the initial state.
- T_A is the set of accepting states such that $T_A \subseteq Q_A$

Each string to be analysed is received as a sequence of symbols of the alphabet, one after the other. Let us assume that A is fully-defined, so for each state there is at least one edge for each symbol in the alphabet. There is a well known and simple technique for modifying any FA to be fully-defined. This consists of adding an additional state and for each symbol in the alphabet an edge (labelled with that symbol) that originates and terminates in this new state. Then edges are added that go from each of the states of the original FA to the new state, until each state is the source of an edge for each symbol. Now let us proceed with the construction of an EXM as follows:

$\Lambda = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T)$ where:

- $X = Z \times Y^*$.
- $Y = \Sigma_A$.
- $Z = \{0, 1\}$.
- The input and output coding functions are of the form:
  $\forall\, s^* \in Y^*, \alpha(s^*) = (g, s^*)$ where $g = 1$ if $q_A \in T_A$ otherwise $g = 0$, and
  $\forall\, g \in Z, \beta(g, <>) = g$
  $\forall\, g \in Z, s^* \in Y^* \mid s^* \neq <>, \beta(g, s^*) = \perp$ where $< >$ denotes an empty input stream

- $Q = Q_A$.
- $I = \{q_A\}$.
- $T = T_A$.
- For each $f_A \in F_A$ with $f_A(q, h) = q'$ in the FA
  $f \in F$ with $f(q, \phi) = q'$ where $\phi \in \Phi$ is such that
  $\phi(g, s^*) = (g', s'^*)$ is guarded by: if $(head(s^*) = h)$
  $s'^* = tail(s^*)$ and
  $g' = 1$ if $q' \in T$, otherwise $g' = 0$

Roughly speaking, the function *head*: $Y^* \rightarrow Y$ returns the first symbol of the string and the function *tail*: $Y^* \rightarrow Y^*$ removes the first symbol of the string. A formal definition of these functions appears below in figure 3. It remains to be shown that A and Λ accept exactly the same language. Let L(A) be the language accepted by the FA and because A is full defined then $\Lambda\uparrow\Phi$.

From the construction of Λ, it is not hard to see that for every string $s^* \in Y^* = \Sigma_A^*$ such that $s^* = h_1::h_2::\ldots::h_n$ there is a path $p_A = (<q_A, q_1, q_2, \ldots, q_n>, < h_1, h_2,\ldots, h_n >)$ in A, and there is a path $p = (<q_A, q_1, q_2, \ldots, q_n>, <\phi_1, \phi_2,\ldots, \phi_n>)$ in Λ where for all $1 \leq i \leq n$, $\phi_i(g_i, s_i^*) = (g_{i+1}, s_{i+1}^*)$ where $head(s_i^*) = h_i$. Now there are two cases:

1. If $q_n \in T_A$ then $q_n \in T$, thus $s^* \in L(A)$ and $g_n = 1$ or
2. If $q_n \notin T_A$ then $q_n \notin T$, therefore $s^* \notin L(A)$ and $g_n = 0$

showing that L(A) is the language accepted by Λ.

□

Lemma 2: For each PDA there exists an EXM that accepts the same language.

Proof. Let $A = (\Sigma_A, \Gamma_A, Q_A, F_A, q_A, T_A)$ be a PDA where:

- Σ_A is the machine's alphabet.
- Γ_A is a finite set of stack symbols.
- Q_A is a finite set of states.
- F_A is a finite collection of transitions of the form $(p, x, r; q, y)$ where:
  $p \in Q_A$ is the current state,
  $x \in \Sigma_A$ is the input symbol,
  $r \in \Gamma_A$ is the symbol popped from the stack,
  $q \in Q_A$ is the new state and
  $y \in \Gamma_A$ is the symbol pushed on the stack.

- The initial state is $q_A \in Q_A$.
- The set of accepting states is $T_A \subseteq Q_A$.

The PDA has an input string and a control mechanism that can be in any of a (finite) number of states. The transitions $(p, x, r; q, y)$ executed by A have the following meaning: In state p, read the symbol x from the input, pop the symbol r from the stack, push the symbol y on the stack, and move to state q. Thus, the current state, the input symbol, and the top of the stack determine the new state and the symbol to be pushed on the stack. The transitions can also refer to the empty string denoted by λ with respect to any of the symbols. For instance $(p, \lambda, r; q, \lambda)$ means that from state p the machine moves to state q while popping the symbol r from the stack. From this an EXM can be constructed as follows:

$\Lambda = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T)$ where:

- $X = Z \times M \times Y^*$.
- $Y = \Sigma_A$.
- $Z = \{0, 1\}$.
- M is a stack of symbols Γ_A with the operations *empty*, *push*, *top* and *pop* defined in the usual manner.
- The input and output coding functions are of the form:
  - $\forall\, s^* \in Y^*, \alpha(s^*) = (g, m_0, s^*)$ where $m_0 = empty(\,)$ and $g = 1$ if $q_A \in T_A$ otherwise $g = 0$, and
  - $\forall\, g \in Z, \beta(g, m, <>) = g$
  - $\forall\, g \in Z, s^* \in Y^* \mid s^* \neq <>, \beta(g, m, s^*) = \perp$ where $<>$ denotes an empty input stream

- $Q = Q_A$.
- $I = \{q_A\}$.
- $T = T_A$.
- For each $(p, x, r; q, y) \in F_A$ in the PDA,
  - $f \in F$ with $f(p, \phi) = q$ where $\phi \in \Phi$ is such that
    - $\phi(g, m, s^*) = (g', m', s'^*)$ where:
      - if $x \neq \lambda$ and $r \neq \lambda$ the function is guarded by the condition: $(head(s^*) = x$ and $top(m) = r)$,
      - if $x = \lambda$ and $r \neq \lambda$ the function is guarded by the condition: $(top(m) = r)$,
      - if $x \neq \lambda$ and $r = \lambda$ the function is guarded by the condition: $(head(s^*) = x)$,
      - if $x = \lambda$ and $r = \lambda$ the function is no guarded by a condition,
      - if $x \neq \lambda$ then $s'^* = tail(s^*)$ else $s'^* = s^*$
      - if $r \neq \lambda$ and $y \neq \lambda$ then $m' = push(y, pop(m))$
      - if $r \neq \lambda$ and $y = \lambda$ then $m' = pop(m)$
      - if $r = \lambda$ and $y \neq \lambda$ then $m' = push(y, m)$
      - if $r = \lambda$ and $y = \lambda$ then $m' = m$
      - $g' = 1$ if $q' \in T$, otherwise $g' = 0$

As in the previous proof, for every path $p_A = (<q_A, q_1, q_2, \ldots, q_n>, <(x_1, r_1, y_1), \ldots, (x_n, r_n, y_n)>$ in A, there is a path $p = (<q_A, q_1, q_2, \ldots, q_n>, <\phi_1, \phi_2, \ldots, \phi_n>)$ in Λ such that for all $1 \leq i \leq n$, $\phi_i(g_i, m_i, s_i^*) = (g_{i+1}, m_{i+1}, s_{i+1}^*)$ is as was described above and as an obvious consequence of this both machines accept the same language.

□

Lemma 3: For each TM there exists an EXM that computes the same relation.

Proof. Let $M = (\Sigma_M, \Gamma_M, Q_M, \delta, q_i, q_h)$ be a TM where:

- Σ_M is a finite set of nonblank symbols.
- Γ_M is a finite set of symbols, such that $\Sigma_M \subseteq \Gamma_M$ and is called the machine's tape symbols.
- Q_M is a finite set of states.
- δ is the transition function of the form $\delta{:}(Q_M' \times \Gamma_M) \rightarrow Q_M \times (\Gamma_M \cup \{Left, Right\})$ where:
  - Q_M' is the set of non-halt states, $Q_M' = Q_M/\{q_h\}$
  - $\{Left, Right\} \notin \Gamma_M$

- The initial state is $q_i \in Q_M$.
- The halt state is $q_h \subseteq Q_M$.

A TM can both read and write on its input (also output) string (*i.e.* tape), so the machine has a *tape head* that can be use to read/write on the tape or can be moved one cell to the right or to the left. Thus,

$\delta(p, x) = (q, y)$ means that when the current state is p and the input symbol is x, the x is replaced by y and the machine moves onto q.

$\delta(p, x) = (q, Left)$ means that from state p when the input symbol is x, the tape head is moved one cell to the left and the control is transferred to state q.

$\delta(p, x) = (q, Right)$ means that from state p when the input symbol x, the tape head is shifted one cell to the right and the machine moves to state q.

Every TM must have at least two states; an initial state q_i from which the machine starts its computation and a halt state q_h with the property that as soon as q_h is reached the machine must stop. An EXM can be constructed as follows:

$\Lambda = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T)$ where:

- $X = Z \times M \times \mathbf{N}^+$.
- $Y = \Gamma_M$.
- $Z = \{non\text{-}halt, halt, abnormal\}$.
- M is an infinite list of symbols from Y.
- The input and output coding functions are of the form:
    - $\forall s^* \in Y^*, \alpha(s^*) = (non\text{-}halt, m_0^*, 1)$ where $m_0^* = s^*$
    - $\forall m^* \in M, i \in \mathbf{N}^+, \beta(halt, m^*, i) = halt$
    - $\forall m^* \in M, i \in \mathbf{N}^+, \beta(abnormal, m^*, i) = abnormal$
    - $\forall m^* \in M, i \in \mathbf{N}^+, \beta(non\text{-}halt, m^*, i) = \perp$

- $Q = Q_A \cup \{q_a\}$.
- $I = \{q_\mathrm{i}\}$.
- $T = \{q_h, q_a\}$.
- For each $\delta(p, x) = (q, y)$ where $p, q \in Q_A$ and $x \in \Gamma_M$ and $y \in \Gamma_M \cup \{Left, Right\}$ in the TM,

    $f \in F$ with $f(p, \phi) = q$ where $\phi \in \Phi$ is such that

    $\phi(g, m^*, i) = (g', m^{*\prime}, j)$ is guarded by the condition: if $(\pi_\mathrm{i}(m^*) = x$ and $i > 0)$ where:

    if $y = Left$ then $j = i - 1$ and $m' = m$

    else if $y = Right$ then $j = i + 1$ and $m' = m$

    else $j = i$ and $\pi_\mathrm{i}(m^{*\prime}) = y$

    $g' = halt$ if $q = q_h$, otherwise $g' = non\text{-}halt$

    $\forall q \in Q \setminus T, f(q, \phi_a) = q_a$ where $\phi_a \in \Phi$ is such that

    $\phi_a(non\text{-}halt, m^*, i) = (abnormal, m^*, i)$ is guarded by the condition: if $(i = 0)$

Every triplet $(g, m^*, i) \in X$ effectively simulates the state of the TM tape and indicates if the halt state has been reached. Let us suppose that the same input is given to both M and Λ, so in every moment m^* contains the same elements in the same order as the tape of M and i indicates the position of the tape head. Additionally, g contains the value *non-halt* until a final state is reached. Because under certain circumstances the computation of a TM may never stop (*i.e.* it never reaches a final state), the decoding function of the EXM is undefined until either q_h or q_a is reached.

An error may occur during the TM computation and this corresponds to the situation when the tape head falls off the left end of the tape and this produces the abnormal termination. This is the reason why in the EXM an additional final state q_a is included, and for each non-terminal state there is a transition from it to q_a, which is triggered whenever, the index value is equal to zero (*i.e.* $i = 0$). This transition changes the value $g = non\text{-}halt$ to $g' = abnormal$ and the computation terminates where the decoding function β can be applied in the obvious manner. Finally, it is not hard to see that for every path $p_M = (<q_i, q_1, q_2, \ldots, q_n, q_h> <\delta_1, \delta_2, \ldots, \delta_h>)$ in M that corresponds to a normal termination, there is a path $p = (<q_i, q_1, q_2, \ldots, q_h>, <\phi_1, \phi_2, \ldots, \phi_h>)$ in Λ, such that m^* and the tape of the TM contains the same symbols in the same order and both machines terminate.

□

3. (Eilenberg) stream X-machines (SXM)

The origins of the SXM are related to the aim to provide the EXM model with capabilities for handling inputs and outputs. Eilenberg [1] suggested an approach to this in which the input and output sets comprise sequences of symbols of some alphabets Σ and Γ. Hence, $Y = \Sigma^*$ is the *input stream*, formed of sequences from the *input* *alphabet* Σ, and Z = Γ* is the *output stream*, formed of sequences from *the output alphabet* and the fundamental data set will have the form X = Γ* x M x Σ* where M is a monoid. The components Γ* and Σ* of the above Cartesian product were respectively called in [25] the *output register* and the *input register*. According to [24] the output sequence can be considered either as instructions to the output device that are interpreted by β, or as a sequence of snap-shots of the output status.

3.1 The SXM model

In this section, the SXM model is studied and the most relevant dissimilarities found in its various definitions are indicated. These differences depend on the main purpose of each investigation, and in general, they do not represent a semantic discrepancy, except for one case. In effect, the normal-termination condition for an SXM establishes that a final control-state has been reached and that the input stream is empty, nevertheless, as we will see, the latter requirement was not originally taken into account. Let us present the original formal description of SXM suggested in [23, 24]:

Definition 13: An EXM with X = Γ* x M x Σ* is an SXM if

- The input and output coding functions have additional properties:
  $\forall\ y \in Y$, $\alpha(y) = (<>, m_0, \alpha^*(y))$ where $<> m_0$ is the initial memory value and α^* is a bijection $\alpha^* : Y \to \Sigma^*$.
  $\forall\ (g^*, m, s^*) \in X$, $\beta(g^*, m, s^*) = \beta^*(g^*)$ where $\beta^* : \Gamma^* \to Z$ is a bijection.

- All the functions remove the head of the input stream and add a symbol to the head of the output stream. Furthermore, no function is allowed to use information from the tail of the input or any symbol of the output. That is to say:
  $\forall\ g^* \in \Gamma^*, m \in M, h \in \Sigma, s^* \in \Sigma^*$,
  if $\exists\ m' \in M, t \in \Gamma$ that depends on m and h then
  $\phi(g^*, m, h{::}s^*) = (t{::}g^*, m, s^*)$ where :: is a concatenation operator.

Convention 2: < > denotes an empty stream.

ADT stream [alphabet ∪ {η}]

Import all from **B** (i.e. Boolean)
Export all

Syntax:
  empty : → stream
  concatenate: alphabet x stream → stream
  is_it_emty: stream → **B**
  head: stream → alphabet ∪ {η}
    (i.e. η is a special symbol that does not belong to the alphabet)
  tail: stream → stream

Axioms:
Let be a ∈ alphabet and let w ∈ stream

  [1] is_it_empty(empty()) = true
  [2] is_it_empty(concatenate(a, w)) = false
  [3] head(empty()) = η (i.e. the special symbol is returned if the stream is empty)
  [4] head(concatenate(a, w)) = a
  [5] tail(empty()) = empty()
  [6] tail(concatenate(a, w)) = w

Figure 2

It is interesting to note that in the above definition the streams operations are carried out with respect to the first symbol (*i.e.* head) of both of them [24]. Subsequently, a definition of SXM was presented a definition in which the symbol of the output alphabet is added at the end [25], and since then this has been adopted in the literature as the standard mode of operation. It should be noted that the position in which the symbol is concatenated in the output stream is immaterial, if the same convention is employed consistently. Let us treat the concept of stream as an abstract data type (ADT for short) as is illustrated in figure 2.

Let ρ: M x $(\Sigma \cup \{\eta\}) \rightarrow \Gamma$, μ : M x $(\Sigma \cup \{\eta\}) \rightarrow$ M and χ : M x $(\Sigma \cup \{\eta\}) \rightarrow \mathbf{B}$, be three partial functions such that:

> $\rho(m, h) \rightarrow t$ produces a new symbol of the output alphabet from *m* and *h*.
> $\mu(m, h) \rightarrow m'$ produces a new memory value *m'* from *m* and *h*.
> $\chi(m, h) \rightarrow b$ produces a Boolean value that depends on *m* and *h*.

From this, the type Φ of the machine can be described alternatively as:

```
If $\Sigma^*$ : stream[$\Sigma \cup \{\eta\}$] and $\Gamma^*$ : stream[$\Gamma$] then
    $\forall g^* \in \Gamma^*, m \in M, s^* \in \Sigma^*$,
    if $(m, \text{head}(s^*)) \in \textit{domain}(\mu)$ and $(m, \text{head}(s^*)) \in \textit{domain}(\rho)$ then
        $\phi(g^*, m, s^*) = (\text{concatenate}(\rho(m, \text{head}(s^*)), g^*), \mu(m, \text{head}(s^*)), \text{tail}(s^*))$; if $\chi(m, \text{head}(s^*))$
    otherwise
        $\phi(g^*, m, s^*) = \perp$
```

This can easily be extended in a similar way to [24] to deal with several cases such as:

$$\phi(g^*, m, s^*) = \begin{cases} (\text{concatenate}(\rho_1(m, \text{head}(s^*)), g^*), \mu_1(m, \text{head}(s^*)), \text{tail}(s^*)) & \text{if } \chi_1(m, \text{head}(s^*)) \\ (\text{concatenate}(\rho_2(m, \text{head}(s^*)), g^*), \mu_2(m, \text{head}(s^*)), \text{tail}(s^*)) & \text{if } \chi_2(m, \text{head}(s^*)) \\ . & \\ . & \\ \perp & \end{cases}$$

where the χ_is are mutually exclusive conditions on *m* and *h*.

The key point to observe about the last definition is that this formalisation of the SXM allows the continuing application of functions, even in the case when the input stream is empty, since it is not explicitly established the contrary. It must be noted that < > has a different meaning from η. The symbol η is returned when the input stream is empty < > and $\eta \notin \Sigma$. In other words, this symbol η defines the head function for the case when the stream is empty.

The definitions of the SXM model found in the literature usually present it as an *n*-tuple, and this report follows this practice, but in the definition that is studied here, the *n*-tuple includes as many as possible of the elements used in the references. As a result, it may appear that definition 14 is redundant (*i.e.* certain components can be derived from others). However, there is a reason for doing this, which is to indicate the meaning of all of these components regardless of whether they are employed or not in a particular reference, or with a particular purpose. Put differently, the ultimate objective of our definition is an illustrative one, and once a better understanding of the elements of an SXM has been gained, a more compact formalisation of the concept (in terms of the size of the *n*-tuple) can be utilised.

Definition 14: An SXM is a 12-tuple $\Lambda = (X, \Sigma, \Gamma, M, \alpha, \beta, Q, \Phi, F, I, T, m_0)$ or alternatively in the particular case when there is only one initial control-state $\Lambda = (X, \Sigma, \Gamma, M, \alpha, \beta, Q, \Phi, F, q_0, T, m_0)$ where:

- The fundamental data set X of the machine is of the form: $X = \Gamma^*$ x M x Σ^*.

- Σ and Γ are respectively the input and output alphabets.

- M is a possible infinite set called the *internal memory* of the machine.

- The input coding function $\alpha : \Sigma^* \rightarrow X$ defines the initial memory-state m_0:
  $\forall s^* \in \Sigma^*, \alpha(s^*) = (< >, m_0, s^*)$

- The output coding function $\beta : X \rightarrow \Gamma^*$ is defined by:
  $\forall g^* \in \Gamma^*, m \in M, \beta(g^*, m, < >) = g^*$
  $\forall g^* \in \Gamma^*, m \in M, s^* \in \Sigma^* \mid s^* \neq < >, \beta(g^*, m, s^*) = \perp$

- Q is the (finite) set of control-states.

- Φ is the type of the machine, a set of non-empty processing relations on X:
  $\Phi: P(X \leftrightarrow X)$

  The type is of the form [22]:
  $\forall g^* \in \Gamma^*, m \in M, \phi(g^*, m, < >) = \bot$
  $\forall g^* \in \Gamma^*, m \in M, h \in \Sigma, s^* \in \Sigma^*,$
  if $\exists\, m' \in M, t \in \Gamma$ that depends on m and h then
  $\phi(g^*, m, h::s^*) = (g^*::t, m', s^*)$ (*i.e.* the output symbol is concatenated at the end)
  otherwise
  $\phi(g^*, m, h::s^*) = \bot$

- F is the next-state (partial) function:
  $F : Q \times \Phi \rightarrow P(Q)$

- I and T are respectively the sets of initial and final states:
  $I \subseteq Q, T \subseteq Q$

  and whenever $I = \{q_0\}$, the control-state q_0 is sometimes written instead of I.

Despite the fact that definitions 13 and 14 provide distinct mechanisms for concatenating symbols with respect to the output, they are really just different ways of modelling the same behaviour. However, the type of the machine is now of the form $\forall g^* \in \Gamma^*, m \in M, \phi(g^*, m, < >) = \bot$, which implies that no function can be computed in an SXM when the input stream is empty (*i.e.* a termination condition). The ADT-stream can be specified to use $\bot$ rather that η, as in figure 3, in order to fit with the properties of definition14.

```
ADT stream [alphabet]

Import all from B
Export all

Syntax:
    empty : → stream
    concatenate: stream x alphabet → stream
    is_it_empty: stream → B
    head: stream → alphabet
    tail: stream → stream

Axioms:
Let be a ∈ alphabet and let w ∈ stream

    [1] is_it_empty(empty()) = true
    [2] is_it_empty(concatenate(w, a)) = false
    [3] head(empty()) = ⊥
    [4] head(concatenate(w, a)) = if is_it_empty(w) then a, else head(w)
    [5] tail(empty()) = ⊥
    [6] tail(concatenate(w, a)) = if is_it_empty(w) then w, else concatenate(tail(w), a)
```

Figure 3

Then, using this ADT-stream (figure 13), the type Φ of the machine can be formalised as:

If Σ^*: stream[Σ] and Γ^*: stream[Γ] then
$\forall g^* \in \Gamma^*, m \in M,$
$\phi(g^*, m, < >) = \bot$

$\forall g^* \in \Gamma^*, m \in M, s^* \in \Sigma^*$

if (m, head(s^*)) $\in$ *domain*(μ) and (m, head(s^*)) $\in$ *domain*(ρ) then

$\phi(g^*, m, s^*)$ = (concatenate(g^*, ρ(m, head(s^*))), μ(m, head(s^*)), tail(s^*)); if χ(m, head(s^*))

otherwise

$\phi(g^*, m, h::s^*) = \perp$

where the χ is a condition on m and h.

In a triplet $(g^*, m, s^*) \in X$ where $g^* \in \Gamma^*$, $m \in M$ and $s^* \in \Sigma^*$, the sequence s^* can be interpreted as the sequence of unprocessed input so far, and g^* as the processed output so far. At this point, the application of ϕ with respect to the operation of the machine consists of:

1. Removing the next input symbol: tail(s*),
2. Updating the memory: μ(m, head(s^*))) and
3. Adding a new output symbol: concatenate(g^*, ρ(m, head(s^*))).

From this, it is easy to see that all the functions ϕ are completely determined by ρ(m, head(s^*)) and μ(m, head(s^*)), so for the sake of simplicity [22, 27] the type of the machine can be referred to as: Φ: M x $\Sigma \rightarrow \Gamma$ x M. That is, instead of writing:

$\phi(g^*, m, h::s^*) = (g^*::t, m', s^*)$ or
$\phi(g^*, m, s^*)$ = (concatenate(g^*, ρ(m, head(s^*))), μ(m, head(s^*)), tail(s^*))

we will respectively write

$\phi(m, h) = (t, m')$ or
ϕ(m, head(s^*)) = (ρ(m, head(s^*))), μ(m, head(s^*))).

The operation of an SXM is similar to an EXM except that in an SXM, the inputs and outputs are streams of symbols. Summarising, in each control-state, a processing function $\phi \in \Phi$ is applied, where the selection depends on the first symbol in the input, the current control-state and the memory. The function then computes the new memory-state (*i.e.* updates the internal memory) and at the same time, the machine generates an output symbol, which is concatenated at the end of the output stream, while the first input symbol is removed from the input stream. The process continues in this way, and the machine traverses a path while generating the stream of outputs until the input stream is empty, and ideally, a final control-state is reached.

Since $X = \Gamma^* \text{ x } M \text{ x } \Sigma^*$, it follows that the fundamental data set can be derived from the alphabets and the memory. Therefore, to avoid redundancy either X or Γ, M and Σ need to be specified in the SXM definition and without loss of generality we choose to employ here the latter form. In the same order of ideas, the initial memory value m_0 can be obtained from the input code function α and *vice versa*. On the other hand, the output code function β is equal to the projection function π_{out} when the input stream is empty, and it is undefined in any other case. In the literature this function has been omitted more in recent papers where m_0 is employed rather than α and the operation of β seems to be implicitly assumed.

Additionally in the literature sometimes Φ is defined as a set of functions *e.g.* in [22, 27, 37], and sometimes Φ is defined as a set of relations *e.g.* in [51, 53, 54], we assume here that Φ is a set of function unless otherwise specified. Table 2 presents a number of equivalent SXM definitions found in the literature.

References	X	Σ	Γ	M	α	β	Q	Φ	F	I	q_0	˥	m_0
[6, 18, 29, 44]		√	√	√			√	√	√		√		√
[4]	√						√	√	√	√		√	
[21, 27, 37, 41, 42, 47, 51]		√	√	√			√	√	√	√		√	√
[22, 23, 24, 25]	√	√	√	√	√	√	√	√	√	√		√	
[53]		√	√	√	√	√	√	√	√		√	√	√
[8, 32]		√	√	√			√	√	√		√	√	√
[34, 45]	√	√	√	√			√	√	√	√		√	√
[54]	√	√	√		√	√	√	√	√	√		√	

Table 2

Having said this, we shall refer to an SXM as either:

> $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, I, T, m_0)$ in the general case, or
> $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, I, m_0)$ if all control-states are terminal, or
> $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, T, m_0)$ when there is only one initial state, or finally
> $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, m_0)$ for the combination of the last two cases

3.2 The SXM computation

As in the EXM, the SXM model has to do with some relation or function *f* on the input and output streams and it is desirable to infer the form taken by *f* from the structure of the SXM. The formalisation of this is given by the next three definitions from [22]:

Definition 15: The *relation computed* f_Λ: $\Sigma^* \leftrightarrow \Gamma^*$ by an SXM Λ is given by:

$$s^* f_\Lambda\, g^* \Leftrightarrow \exists\, q_0 \in I,\, q \in T \text{ and } p: q_0 \rightarrow q \text{ such that } \pi_{\text{out}}(|p|(<>, m_0, s^*) = g^*$$

This obviously defines the output (or outputs) corresponding to an input when the machine follows a successful path. This preliminary idea must suffice for now, but we shall take it up again later in section 4.3 to devise concepts that are more detailed (*i.e.* fully-computable, weak stream function, stream function).

Definition 16: The *transition-relation* $[\Lambda]_0$ of an SXM Λ can be described by:

$$[\Lambda]_0: Q \times M \times \Sigma \leftrightarrow \Gamma \times Q \times M,$$
$$(q, m, h)\, [\Lambda]_0\, (t, q', m') \Leftrightarrow \exists\, \phi: q \rightarrow q' \text{ such that } \phi(m, h) = (t, m')$$

The transition-relation describes all the transitions of the machine Λ. Therefore, the following relation can define the entire computation of the SXM:

Definition 17: The *extended-transition-relation* $[\Lambda]$ of an SXM Λ is given by:

$$[\Lambda]: Q \times M \times \Sigma^* \leftrightarrow \Gamma^* \times Q \times M \text{ with:}$$
$$\forall\, q \in Q,\, m \in M$$
$$(q, m, <>)\, [\Lambda]\, (<>, q, m')$$

$$\forall\, q, q' \in Q,\, m, m' \in M,\, h \in \Sigma,\, s^* \in \Sigma^*,\, t \in \Gamma,\, g^* \in \Gamma^*$$
$$(q, m, h{::}s^*)\, [\Lambda]\, (t{::}g^*, q', m') \Leftrightarrow \exists\, q'' \in Q,\, m'' \in M \text{ where } (q, m, h)\, [\Lambda]_0\, (t, q'', m'') \text{ and } (q'', m'', s^*)\, [\Lambda]\, (g^*, q', m')$$

In [37] the concept of the data set is widened to the concept of configuration, which includes the same elements as the data set X but also the current control-state.

Definition 18: A configuration of an SXM is a 4-tuple (m, q, s^*, g^*) where $m \in M$, $q \in Q$, $s^* \in \Sigma^*$ and $g^* \in \Gamma^*$. An SXM starts its execution from an initial configuration having the form $(m_0, q_0, s^*, <>)$ where $q_0 \in I$, and s^* is the initial input sequence. A change of configuration denoted by

> $(m, q, s^*, g^*) \vdash (m', q', s'^*, g'^*)$ is possible if there is a function $\phi(m, h) = (t, m') \in \Phi$ such that $q' \in F(q, \phi)$, $\text{head}(s^*) = h$, $\text{tail}(s^*) = s'^*$, $g'^* = \text{concatenate}(g^*, t)$ and therefore $\rho(m, \text{head}(s^*)) = t$ and $\mu(m, \text{head}(s^*)) = m'$.

The transitive and reflexive closure of $\vdash$ is denoted by $\vdash^*$. A configuration $(m, q, <>, g^*)$ is final if the input stream is empty and $q \in T$.

It is interesting to observe the equivalence among the concepts of configuration and change of configuration proposed in [37], with the concepts of transition-relation and extended-transition-relation suggested in [22]. We show this by the following two propositions.

Proposition 4: $(m, q, s^*, g^*) \vdash (m', q', s'^*, g'^*) \Leftrightarrow (q, m, h)\, [\Lambda]_0\, (t, q', m')$.

Proof. From definition 18, $(m, q, s^*, g^*) \vdash (m', q', s'^*, g'^*)$ then $\exists\ \phi(m, h) = (t, m') \in \Phi$ with $q' \in F(q, \phi)$, $\text{head}(s^*) = h$, $\text{tail}(s^*) = s'^*$, $g'^* = \text{concatenate}(g^*, t)$ and $\rho(m, \text{head}(s^*)) = t$, $\mu(m, \text{head}(s^*)) = m'$ and from definition 16, $(q, m, h)\ [\Lambda]_0\ (t, q', m') \Leftrightarrow \exists\ \phi: q \rightarrow q'$ such that $\phi(m, h) = (t, m')$.

□

Proposition 5: $(m, q, s^*, g^*) \vdash^* (m', q', s'^*, g'^*) \Leftrightarrow (q, m, s^*)\ [\Lambda]\ (g'^*, q', m')$.

Proof. It is easy to see that $(m, q, s^*, g^*) \vdash^* (m', q', s'^*, g'^*)$ implies that there is a path p from state q to state q' such that $|p|(g^*, m, s^*) = (s'^*, m', g'^*)$ this implies that $(q, m, s^*)\ [\Lambda]\ (g'^*, q', m')$. The other direction of the demonstration follows the same arguments.

□

The original intention behind definition 18 was to cope with the possibility of providing communication capabilities to the model of SXM and it offers a natural way to derive the input-output relationship computed by an SXM, that is the output corresponding to an input sequence.

Definition 19: The output corresponding to an input sequence $s^* \in \Sigma^*$ computed by an SXM Λ is:

$$\Lambda(s^*) = \{g^* \in \Gamma^* \mid \exists\ m \in M, q_0 \in I, q \in T, \text{ so that } (m_0, q_0, s^*, <>) \vdash^* (m, q, <>, g^*)\}.$$

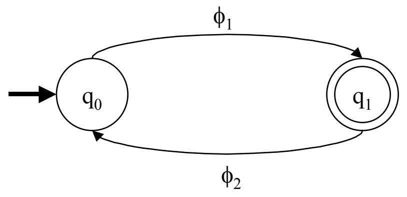
A very interesting observation arises here with respect to the above concepts that we formulate as:

Corollary 1: For any SXM Λ and for all $s^* \in \Sigma^*$, $f_\Lambda(s^*) = \Lambda(s^*)$.

Proof. From definitions 15,19 and proposition 5.

□

Example 4: We again consider the computation of the function of example 1, but now let us specify it by an SXM. The specification of this machine is presented in figure 4. The machine operates in a loop composed by two functions, the function ϕ_1 reads and stores the first element of a pair (a_i, b_i). Function ϕ_2 reads the second element, and computes the multiplication and the summation.



$\Phi = \{\phi_1, \phi_2\}$ $\phi_1(m, h) = (t, m')$ where: $t = h$ $m'.a = h$ $m'.r = m.r$ $\phi_2(m, h) = (t, m')$ where: $t = m'.r$ $m'.a = m.a$ $m'.r = (h * m.a) + m.r$	$Q = \{q_0, q_1\}$ $I = \{q_0\}$ and $T = \{q_1\}$ $F = \{f_1, f_2\}$ where: $f_1(q_0, \phi_1) = q_1$ $f_2(q_1, \phi_2) = q_2$ Initial memory m_0 $m_0.a = 0$ $m_0.b = 0$

Figure 4

Let us suppose that the input is organised as $a_1, b_1, a_2, b_2, \ldots, a_n, b_n$ where each element corresponds to a symbol, since that $\Sigma = \mathbf{N}^+$. Because in an SXM an output is required with each function performed, whenever an element of the first vector is read (*i.e.* each a_i) the same symbol is written and when an element of the second vector is read (*i.e.* each b_i) the partial result computed by the machine until that moment is displayed, and therefore $\Gamma =$

$\mathbf{N}^+$. The memory has two components, thus M = **N** x **N**, and we denoted them respectively *m.a* and *m.r* where *m.a* is used to store the elements of the first vector and *m.r* keeps the result of the computation, from this $m_0.a = m_0.r = 0$. The specification of this machine is presented in figure 4.

Clearly, the paths that the machine follows are of the form $p = (<q_0, q_1, q_0, \ldots, q_0> <\phi_1, \phi_2, \ldots, \phi_1, \phi_2>)$ with $|p_Q| = 2^n + 1$ and $|p_\Phi| = 2^n$ where *n* is the cardinality of the vectors. Taking the problem instance $v_1 = <2, 3, 4>$ and $v_2 = <5, 1, 7>$ then the input stream must be $s^* = 2::5::3::1::4::7$ and $f_\Lambda(2::5::3::1::4::7) = (2::10::3::13::4::34)$ is in the relation computed by the SXM. Additionally, $(q_0, m_0, 2)\ [\Lambda]_0\ (2, q_1, m_1)$ and $(q_1, m_1, 5)\ [\Lambda]_0\ (10, q_0, m_2)$ where $m_1.a = 2$, $m_1.b = 0$, $m_2.a = 2$ and $m_1.b = 10$ are in the transition-relation of the machine, and it follows that $(q_0, m_0, 2::5)\ [\Lambda]\ (2::10, q_0, m_2)$ is in the extended-transition-relation.

From definition 18, and propositions 4 and 5 it follows that $(m_0, q_0, 2::5::3::1::4::7, < >) \vdash (m_1, q_1, 5::3::1::4::7, 2)$ and $(m_0, q_0, 2::5::3::1::4::7, < >) \vdash^* (m_2, q_0, 3::1::4::7, 2::10)$. Because $(m_0, q_0, 2::5::3::1::4::7, < >) \vdash^* (m_f, q_0, < >, 2::10::3::13::4::34)$ where by definition 19, $\{2::10::3::13::4::34\} \in \Lambda(2::5::3::1::4::7)$.

■

3.3 Conformance Testing and SXM

The objective of testing is to determine whether an IUT conforms (is equivalent) to its specification. Clearly, this depends greatly on the conformance relation that is assumed with respect to the validity of an IUT (*i.e.* what a valid IUT is, and under which circumstances the IUT is valid).

In general, the conformance behavioural (*i.e.* black-box or functional) testing problem (also called fault detection problem) can be described as follows. Taking a specification and an IUT, if the latter is given as a black-box then only its input/output behaviour can be observed, and consequently the testing process consists of comparing this observable behaviour against the observable behaviour of the specification. During testing inputs are sent to and outputs are received from the IUT, then these outputs are compared with the expected outputs of the specification. From the execution of this process, a conclusion about the conformance relation can be obtained, and any sequence that solves (at least partially) this problem is called a *test sequence*. Various approaches have been presented for conformance black-box testing generation, which are mainly based on automata theory. Therefore, these approaches usually focus on FSM, which is defined as follows:

Definition 20: A FSM is a 5-tuple $K = (\Sigma, \Gamma, Q, \varphi, \kappa)$ where:

- Σ and Γ are two finite non-empty sets of input and output symbols respectively.
- Q is finite set of states.
- φ is the state transition function:
  $\varphi: Q \times \Sigma \rightarrow Q$
- κ is the output function:
  $\kappa: Q \times \Sigma \rightarrow \Gamma$

The SXMT is a behavioural (black-box) testing technique based on the Chow W-method [55] and Fujiwara Wp-method [56], in which two arbitrary SXM representing respectively, the specification and the IUT are compared in order to analyse if they compute the same function. Furthermore it attempts to identify the same type of faults the FSM testing algorithms usually tried to find, that is [22]:

- missing states
- extra states
- missing transitions
- extra transitions
- mis-directed transitions
- transitions with faulty functions (input-output).

An important remark is that Holcombe and Ipate [22, 25, 27] based their method on a *reductionist* approach for generating a finite test set but taking into account that M could be infinite. In effect, they proposed that the solution of the test problem for a complete system could be reduced to the solution of the test problem of the system's components. The technique used for this depends only on the control-states, and is independent of the different possible memory-states. Therefore, it produces a much smaller test set than would be possible with any method that had to consider both sets of states. Furthermore, the process of constructing the test set also identifies the conditions that need to be fulfilled [22, 27]. If these constraints are satisfied, it has been demonstrated that the test set which is generated corroborates the functional equivalence between SXM behaviour (*i.e.* find all faults) [27]. Let us suppose an implementation ϑ will be tested against an SXM specification Λ, the standard SXMT made the following assumptions for the test to be at all possible [22, 31]:

1. Λ has the same set Φ of functions as ϑ
2. There is a known integer k such that
   $|Q'| - |Q| \leq k$ where
   $|Q'|$ and $|Q|$ respectively the number of states in the implementation and the specification.

3. Λ and ϑ are *deterministic* SXM.
4. Λ is *reachable*.
5. The set of functions Φ is *complete* with respect to M.
6. The set of functions Φ is *output-distinguishable*.
7. The associated automata of Λ and ϑ are *minimal*.

Assumption 1: The method takes for granted that the same set of processing functions are used by both the implementation and specification, thus it is assumed that the implementation of these functions is correct (*i.e. reductionist* approach). That is to say, if the system, an SXM Λ, is composed of the parts $\phi_1, \phi_2, \dots, \phi_n \in \Phi$, then by applying the SXMT on Λ, it is possible to deduce that Λ is fault-free if Φ is fault-free. A significant advantage is that the same approach can be applied to each component of the systems if each $\phi \in \Phi$ is specified as an SXM or alternatively the functions can be tested using some other technique [22].

Assumption 2: This condition places an upper boundary on the number of states in the implementation; otherwise, it would be impossible to establish how long a test sequence has to be in order to reach all the states. Instead, this assumption can be expressed as; the faults may increase the number of states in the implementation, but this number is always known (or at least a good estimation can be done). Evidently, if $k > 0$ then the conformance testing problem becomes harder since it implies a search in an unknown state-transition diagram of k states. For FSM Vasilevskii [57] (reported in [58]) has demonstrated that for k extra nodes the lower boundary on the test sequence is multiplied by $|\Sigma|^K$ where $|\Sigma|$ is the number of input symbols of the FMS. For the SXMT, this dependence on k is $|\Phi|^k$.

The rest of the conditions listed above, are reviewed in the following four sections.

3.4 Determinism and SXM

Determinism has played a very important role in the theory of EXM since the SXMT was originally developed to deal only with deterministic machines [22, 25, 27, 28]. However, it has recently been modified to deal with non-determinism [31, 32, 33, 34, 35, 36]. Originally, the definition of determinism separated the set of all SXM by establishing that those machines that did not fit in that definition were considered non-deterministic. Later in order to extend the testing method Hierons *et al.* [31] identified a form of non-determinism called *quasi-non-determinism*. Since the objective in this report is to present the basis of the EXM-theory with relation to testing, these concepts are reviewed below.

There are three sources of non-determinism for a SXM. In principle, it is possible that the cardinality of the set of initial states could be greater than one. Nevertheless, following [32] this does not affect the test problem, since a test sequence may be obtained for each possible initial state and then all of them combined at the end. For the sake of simplicity the definition of a SXM with respect to determinism, includes only one initial state q_0. The second form of non-determinism, called *state non-determinism* in [31] refers to the fact that from a particular state the application of a processing function could result in more than one possible state.

Definition 21: A SXM $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, T, m_0)$ has state non-determinism if

$$\exists\, q \in Q, \phi \in \Phi \text{ with } (q, \phi) \in \mathit{domain}(F) \text{ such that } |F(q, \phi)| > 1.$$

It must be clear that if an SXM has state non-determinism, then its associated FA is a non-deterministic (NFA). From this it follows that the same standard techniques for transforming a NFA into a deterministic one (DFA) can be applied in a straightforward manner to SXM. Therefore, state non-determinism can be removed without affecting the relation computed by the SXM, but at the cost of a less compact machine.

The third source of non-determinism called *operator non-determinism* in [31], occurs when some (may be all but at least one) of the elements of Φ are relations.

Definition 22: A SXM $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, T, m_0)$ has operator non-determinism if

$$\exists\, m \in M, h \in \Sigma, \phi \in \Phi \text{ with } (m, h) \in domain(\phi) \text{ such that } |\phi(m, h)| > 1.$$

For obvious reasons operator non-determinism cannot be removed by rewriting the SXM, as is the case with state non-determinism.

Definition 23: A SXM $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, T, m_0)$ is called quasi-non-deterministic [31], denoted by QNSXM if $\forall\, \phi, \phi' \in \Phi$ with $\phi \neq \phi'$, emerging from the same state then $domain(\phi) \cap domain(\phi') = \emptyset$.

It is clear that the above definition establishes that any two processing functions (relations) that can be applied from a particular state must have disjoint domains.

As should be expected the concept of determinism for SXM is directly derived from DEXM (definitions 7 and 8) and it does not allow the state non-determinism, the operator non-determinism, and the intersection of the domains of the functions emerging from the same state.

Definition 24: A SXM $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, T, m_0)$ is called deterministic, denoted by DSXM [27] if

- The set of initial states contains only one element q_0.
- $F: Q \times \Phi \rightarrow Q$
- $\Phi: P(X \rightarrow X)$ (*i.e.* Φ contains only functions).
- $\forall\, \phi, \phi' \in \Phi$ with $\phi \neq \phi'$, emerging from the same state then $domain(\phi) \cap domain(\phi') = \emptyset$.

Hence, a DSXM Λ computes a (partial) function $f_\Lambda: \Sigma^* \rightarrow \Gamma^*$. Finally, for completeness we shall say that a SXM that is neither a DSXM nor a QNSXM is a non-deterministic SXM (NSXM).

3.5 Reachability and attainability of SXM

Informally speaking, reachability is the property that answers the question what can be done (*i.e.* situations reached) in a system? In other words, what are the states to which the system can get from an initial state, and what states are inaccessible?

Definition 25: A state $q \in Q$ is *reachable* in a SXM if

$$\exists\, p: q_0 \rightarrow q, s^* \in \Sigma^*, g^* \in \Gamma^* \text{ such that } |p|(<>, m_0, s^*) = (g^*, m, s'^*) \text{ where } q_0 \in I.$$

A SXM in which all states are reachable is called a *reachable*-SXM (*r*-SXM). Let us define $reach(q)$ = true, if q is a reachable state, otherwise $reach(q)$ = false. From this a SXM is a *r*-SXM, if $\forall\, q \in Q$, $reach(q)$ = true. It is easy to see that if in a SXM some states are not reachable they can be removed, together with all the edges related to them, without affecting the function computed by the SXM. Thus, any SXM that is not reachable can be transformed directly into a *r*-SXM by taking into account only the reachable part of it. From now on, we will consider only those machines, which are *r*-SXM referring to them simply as SXM.

On the other hand, the set of memory values that can be computed by a SXM in a given state are said to be *attainable* in that state, formally:

Definition 26: If $\exists\, p: q_0 \rightarrow q, s^* \in \Sigma^*, g^* \in \Gamma^*$ such that $\pi_{mem}(|p|(<>, m_0, s^*))$ is defined where $q_0 \in I$ then the memory value $m = \pi_{mem}(|p|(<>, m_0, s^*))$ is called *attainable* in q and $attain(q, m)$ = true, if $m \in M$ is attainable in q, otherwise $attain(q, m)$ = false.

The set of memory values that can be taken when the SXM Λ is in state q can be defined as:

Definition 27: For all $q \in Q$,

$Mattain_q(\Lambda) = \{m \in M$ such that $attain(q, m) = \text{true}\}$ or alternatively
$Mattain_q(\Lambda) = \{m \in M \mid \exists\, q_0 \in I, s \in \Sigma^*, g \in \Gamma^* \mid (m_0, q_0, s, <>) \vdash^* (m, q, <>, g)\}$.

A memory value $m \in M$ is *ϕ-attainable* if m is attainable in at least one state from which ϕ emerges. $Mattain_\phi(\Lambda)$ denotes the set of all the memory values that are ϕ-attainable for a particular ϕ or rather;

Definition 28: For any $\phi \in \Phi$, $m \in M$ is *ϕ-attainable* if $\exists\, q \in Q$ such that $F(q, \phi) \neq \varnothing$, $m \in Mattain_q(\Lambda)$ and

$$Mattain_\phi(\Lambda) = \bigcup_{\exists q \in Q \mid F(q,\phi) \neq \varnothing} Mattain_q(\Lambda)$$

This leads to the general definition of attainability of the memory.

Definition 29: The *attainable* memory of a SXM Λ, is:

$$Mattain(\Lambda) = \bigcup_{\forall q \in Q} Mattain_q(\Lambda)$$

Clearly, for any reachable state $q \in Q$ of a SXM Λ, it follows from the previous definitions that $Mattain_q(\Lambda) \neq \varnothing$ and since we are considering only *r*-SMX this is true for all $q \in Q$ implying that also $Mattain(\Lambda) \neq \varnothing$. Because we are considering SXM for which for all $\phi \in \Phi$ there is at least one edge in the machine and at least one $q \in Q$ such that $F(q, \phi) \neq \varnothing$, then:

Corollary 2: $Mattain(\Lambda) = \bigcup_{\phi \in \Phi} Mattain_\phi(\Lambda)$ therefore $\bigcup_{\forall q \in Q} Mattain_q(\Lambda) = \bigcup_{\phi \in \Phi} Mattain_\phi(\Lambda)$

Proof. From definitions 27 and 29.

□

3.6 Design-for-test conditions: Completeness and output-distinguishability

The SXMT method verifies whether two SXM implement the same input-output relationship [27]. In order to do this, certain constraints are imposed on the design to ensure the correct and complete generation of the test set. The idea of design for test (*i.e.* the design should be restricted directly by the specification) is not new and has been used in hardware design for a long time. The design for test requirements impose limits on the designer's choices on how to use a particular formalism, so as to avoid a product being released that might be very hard or impossible to test. There is no reason why the same view of the specification as a *test-centred* document with design constraints cannot be handled in software, if that ensures a reliable testing technique [22]. These constraints, called the *design-for-test* conditions, can be formulated as follows:

Condition 1, Completeness: In general, the type Φ of a SXM is *complete* with respect to the memory M, if it is always possible to apply an input that can trigger a particular function (relation) as long as the current memory value is known. This ensures that any path of the SXM can be followed from the initial memory value and from the initial control-state.

Condition 2, *Output-distinguishability*: If from any output symbol produced by a SXM with a given memory value, it is possible to determine which function (relation) has been applied, then the type Φ is *output-distinguishable*.

The notions of completeness and output-distinguishability are formalised in definition 31 and 33 but in order to get a better understanding of them, it is necessary to establish first what is meant that the type of a machine is *closed*. For the following, given a SXM $\Lambda = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$, let $M' \subseteq M$ be a subset of the memory and let $S = \{S^\phi \mid \phi \in \Phi\}$ be a family of non-empty subsets of Σ, $S^\phi \subseteq \Sigma$ index by ϕ. Hence, we arrive at our next definition [47]:

Definition 30: Φ is *closed* with respect to (M', S) denoted by $\Phi \otimes (M', S)$, if

- $m_0 \in M'$ and

- $\forall \phi \in \Phi, \forall m \in M', \forall h \in S^{\phi}$, if $\phi(m, h) = (t, m')$ with $t \in \Gamma$ then $m \in M'$

Expressed in words, if the application of any function with a memory value from a subset of the memory M', and an input symbol from the subsets of the input alphabet S^{ϕ}, always produces a memory value of the same subset M' of the memory then the type Φ is closed.

Completeness establishes that it is always possible to apply an input *h* that is able to trigger a particular function φ as long as the current memory *m* value is known. In other words, every processing function φ will be capable of processing all memory values of M' using inputs from S^{ϕ}. Formally [32, 47]:

Definition 31: Φ is *complete* with respect to (M', S) denoted by Φ^{C} (M', S), if

- $\Phi \otimes (M', S)$ and
- $\forall \phi \in \Phi, \forall m \in M', \exists h \in S^{\phi} \mid \phi(m, h) \neq \varnothing$ (or equivalently $(m, h) \in domain(\phi)$),

Moreover, if $M' = M$ and $\forall \phi \in \Phi, S^{\phi} = \Sigma$ then Φ is called *complete* and is denoted Φ^{C}.

On account of the fact that completeness guarantees that any path of the associated FA of an SXM can be followed from the initial state and the initial memory value, it is important to note that the notion of completeness established in definition 11 refers to different properties. The notion of $\Lambda\uparrow X$ in the context of SXM can be expressed as in [34]:

Definition 32: An SXM Λ is *completely-specified*, denoted by $\Lambda\uparrow(\Sigma, M)$ if $\forall h \in \Sigma, m \in M, q \in Q, \exists \phi \in \Phi$ such that $(q, \phi) \in domain(F)$ and $\phi(m, h) \neq \varnothing$.

The idea behind the above definition (and also definition 11) is that for every state and for every memory-input pair there exists a function that can be applied to them; otherwise, the machine could arrive at a certain state in which no transition would be applicable. Evidently, every EXM Λ can be modified into an equivalent EXM Λ' which is completely specified. The procedure is similar to that used in automata theory (*i.e.* the one employed in lemma 1) to obtain a transition function which is fully-defined [52]. Basically, an extra state q_e is included in Λ', then for each pair (*m*, *h*) we shall define $\phi(m, h) = (m, h)$ with $F(q_e, \phi) = q_e$. All unspecified transitions for each state *q* of Λ are added in Λ' in the form of edges from *q* to q_e.

In its most general form, the output-distinguishability property with respect to Φ states that it is possible to determine from the observation of the output which processing function was applied. That is, if the memory value is known the application of one input will produce an output that cannot be produced for two or more different functions. From [22, 25] we have the following:

Definition 33: Φ is *output-distinguishable* with respect to (M', S) denoted by Φ^{Γ} (M', S), if

- $\Phi \otimes (M', S)$ and
- $\forall \phi_i, \phi_j \in \Phi, \forall m, m'_1, m'_2 \in M', \forall h \in (S^{\phi i} \cap domain(\phi_j)) \cup (S^{\phi j} \cap domain(\phi_i)) \mid \phi_i(m, h) = (t, m'_1)$ and $\phi_j(m, h) = (t, m'_2)$ with $t \in \Gamma \Rightarrow \phi_i = \phi_j$.

Further, if $M' = M$ and $\forall \phi \in \Phi, S^{\phi} = \Sigma$ then Φ is called *output-distinguishable*. This is denoted Φ^{Γ}.

In this respect, Hierons *et al.* [31, 32] and Ipate *et al.* [34] have remarked (when working on NSXM) on an interesting property. If Φ is a set of relations with Φ^{Γ} although we can identify the φ executed by the machine from the input provided and the output observed, the current state of the memory will be unknown. So to deal with non-determinism either; the definition of output-distinguishability must be augmented as in definition 34 [34] or an additional property can be identified as in definition 35.

Definition 34: if Φ is a set of relations then it is *r-output-distinguishable* with respect to (M', S) denoted by $\Phi^{R\Gamma}$ (M', S), if

- $\Phi \otimes (M', S)$ and
- $\forall \phi_i, \phi_j \in \Phi, \forall m, m'_1, m'_2 \in M', \forall h \in (S^{\phi i} \cap domain(\phi_j)) \cup (S^{\phi j} \cap domain(\phi_i)) \mid \phi_i(m, h) = (t, m'_1)$ and $\phi_j(m, h) = (t, m'_2)$ with $t \in \Gamma \Rightarrow \phi_i = \phi_j$ and $m'_1 = m'_2$

If M' = M and $\forall \phi \in \Phi$, $S^{\phi} = \Sigma$ then Φ is called *r-output-distinguishable*, denoted by $\Phi^{\Gamma R}$.

On the other hand, according to [31, 32] the type Φ is observable when knowing the current memory value it is possible to deduce the next memory value from the observed input/output behaviour of the machine. That is to say, when an input is provided to the machine, the next memory value may differ only if the output observed differs.

Definition 35: Φ is *observable* with respect to (M', S) denoted by Φ^{O} (M', S), if

- $\Phi \otimes$ (M', S) and
- $\forall \phi \in \Phi, \forall m \in M', \forall h \in S^{\phi} \mid \{(t', m'), (t'', m'')\} \in \phi(m, h)$, if $(t' = t'') \Rightarrow (m' = m'')$

Moreover, if M' = M and $\forall \phi \in \Phi$, $S^{\phi} = \Sigma$ then Φ is called *observable* and is denoted Φ^{O}.

Let us analyse the last two properties in more detail, clearly, if Φ is a set of (partial) functions, it is easy to see that Φ^{O} but not necessarily Φ^{Γ}. If Φ^{Γ} this implies that for any pair of functions $\phi_1(m, h) = (t', m')$ and $\phi_2(m, h) = (t'', m'')$ if $t' = t''$ then $\phi_1 = \phi_2$ and trivially $m' = m''$, otherwise (*i.e.* $t' \neq t''$) $\phi_1 \neq \phi_2$, in any case the next memory value is known. Obviously, Φ^{Γ} also implies that Φ^{O}.

Now consider the case when Φ is a set of relations. If Φ^{Γ} (as in definition 33) for any pair $\phi_1(m, h) = (t', m')$ and $\phi_2(m, h) = (t'', m'')$, $t' = t''$ implies that $\phi_1 = \phi_2$, nevertheless the next memory state is unknown. Hence, Φ^{Γ} does not necessarily imply that Φ^{O}. Going further, if Φ^{Γ} and Φ^{O} for the same pair ϕ_1 and ϕ_2 we have that $t' = t''$ implies that $\phi_1 = \phi_2$ and $m' = m''$ therefore $\Phi^{\Gamma R}$ (as in definition 34).

n.b. if the outputs differ (*i.e.* $t' \neq t''$) this does not mean that $\phi_1 \neq \phi_2$, so it cannot be determined by the simple observation of two different outputs neither the relation that was applied and the memory value.

In what follows, we will refer to the concept of definition 34 simply as output-distinguishability and it will be denoted by Φ^{Γ}. Following [47], Φ^{Γ} and Φ^{C} can be fulfilled less restrictively (*relaxed*) by taking into account only the subset of memory that can be reached from an initial state.

Definition 36: Φ is *input-complete* with respect to (M', S) denoted by $\Phi^{\Sigma C}$ (M', S), if

- $\Phi \otimes$ (M', S) and
- $\forall \phi \in \Phi, \forall m \in Mattain_{\phi}(\Lambda) \cap M', \exists h \in S^{\phi} \mid \phi(m, h) \neq \emptyset$

Overall, if M' = M and $\forall \phi \in \Phi$, $S^{\phi} = \Sigma$ then Φ is called *input-complete* and this is denoted $\Phi^{\Sigma C}$.

Definition 37: Φ^{Γ} (M', S), if

- $\Phi \otimes$ (M', S) and
- $\forall \phi_i, \phi_j \in \Phi, \forall m \in Mattain_{\phi 1}(\Lambda) \cap Mattain_{\phi 2}(\Lambda), \forall m'_1, m'_2 \in M', \forall h \in (S^{\phi i} \cap domain(\phi_j)) \cup (S^{\phi j} \cap domain(\phi_i)) \mid \phi_i(m, h) = (t, m'_1)$ and $\phi_j(m, h) = (t, m'_2)$ with $t \in \Gamma \Rightarrow \phi_i = \phi_j$ and $m'_1 = m'_2$,

Taken as a whole, if M' = M and $\forall \phi \in \Phi$, $S^{\phi} = \Sigma$ then Φ^{Γ},

Obviously, definitions 36 and 37 are particular cases of definitions 31 and 33, where only memory values that are ϕ-attainable are taken into consideration. In the SXMT, completeness and output-distinguishability are conditions that need to be fulfilled (or at least their relaxed variants) by the specification SXM, mainly because the SXMT tests if two SXM (*i.e.* specification and IUT) compute the same function. It does this by testing if their associated FA are equivalent [22]. In the literature, these two prerequisites are called *design-for-test conditions*, since without them it is very difficult (if not impossible), to test a system using this SXMT.

These design-for-test conditions can be introduced into a specification by extending the definition of the function of the type Φ of the machine. This rewriting of the SXM might involve the addition of new input and output symbols, which of course can be removed when testing is done.

If Φ is not Φ^{C} then for every $\phi \in \Phi$ which is not complete an input $e \notin \Sigma$ is included for testing purposes. It would seem reasonable to define a new ϕ_e as:

$\phi_e(m, h) = \phi(m, h)$, if $(m, h) \in domain(\phi)$
$\phi_e(m, h) = (t, m)$, if $h = e$ and $m \in$ M where $t \in \Gamma$ is selected arbitrarily.

Thus, ϕ_e has the same M as the original machine and its input alphabet is $\Sigma \cup \{e\}$. This approach will lead naturally to a new type for the machine Φ_e. On the basis of determinism, for any two ϕ_e and ϕ_e' that emerge from the same state it will suffice that the new extra testing symbols employed for each one of them be different. Then by the definition 31, Φ_e is Φ_e^C.

The corresponding modification of an SXM with respect to the output-distinguishability property is achieved by augmenting the output alphabet. Let Φ be not Φ^Γ then by defining $\Gamma' = \Gamma$ x $\{1, 2, \ldots, k\}$ where $k = |\Phi|$ and $o: \Phi \to \{1, 2, \ldots, k\}$ is a one-to-one mapping. Thus, using this convention for every $\phi \in \Phi$ we may define a new ϕ_e as:

if $\phi(m, h) = (t, m)$ then $\phi_e(m, h) = ((t, o(\phi)), m)$ and therefore by definition 33, Φ_e is Φ_e^Γ.

3.7 The minimal associated FA of an SXM

The associated FA of an SXM is a highly important concept in the SXMT, since one of the main results [27] demonstrates that two functions computed by two SXM are equal if the two associated FA of these two SXM are isomorphic. In addition, in the SXMT the minimality of the associated FA ensures the existence of some basic sets (*i.e.* characterisation set, state cover and transition cover), needed to perform the method and to corroborate, if it exists, the isomorphism. The central idea behind the minimisation of a DFA, is that there is a DFA that has fewer states than any other DFA that accepts the same language and is called minimal (or minimum-state equivalent) DFA. The notion of state equivalence is fundamental in order to achieve this minimisation, because when a number of states are equivalent they can be replaced by one state that behaves like all of them.

Definition 38: Let A = (Φ, Q, F, q_0, T) be the associated DFA of a DSXM and let $S \subseteq \Phi^*$. If $q, q' \in Q$ such that $\forall\ s^* \in S$, there are two paths p and p' starting respectively from q and q' and the final state of both of them is either terminal or non-terminal then q and q' are *S-equivalent*, if not they are *S-distinguishable*. Moreover if S = Φ^* then q and q' are called *equivalent* for the first case, and *distinguishable* otherwise.

In the SXMT to ensure that during testing all the outputs (including those that correspond to intermediate computations) can be observed, it is assumed that T = Q (*i.e.* all states are terminal). Clearly, this feature can be removed once the testing has been done. If this is the case, we shall say that:

Definition 39: Let A = (Φ, Q, F, q_0, T) be the associated DFA of a DSXM with T = Q, and let $S \subseteq \Phi^*$. If $q, q' \in Q$ such that $\forall\ s^* \in S$, there are two paths p and p' starting respectively from q and q' then q and q' are *S-equivalent*, if not they are *S-distinguishable*. As in definition 38, if S = Φ^* these states are simply called *equivalent* or *distinguishable*.

The function below is a modification of the table-filled algorithm presented in [59] that finds all the pairs of states that are equivalent, and those that are not. The difference is that the version here solves the same problem in the case when T = Q.

```
Let $\theta$: Q x Q $\to$ Boolean be a function of the form:
$\theta(q, q') = b$ where:
    if $(\forall\ s \in \Phi, (q, s) \notin domain(F) \wedge (q', s) \notin domain(F)) \vee (\forall\ s \in \Phi$, equivalence$(F(q, s), F(q', s), F, \Phi))$
    then $b = true$
    otherwise $b = false$
```

Obviously, the basis of the function occurs when there are no paths (except for the empty path) emerging from q or q', then the pair (q, q') is equivalent. The induction is as follows: For any pair of states q and q' and for each input symbol. If there is no edge emerging from either of them that is labelled with the input symbol, or if such an edge exists it goes from q and q' to a pair of equivalent states then they are equivalent. If two states are not equivalent then they are distinguishable. However, more important is that:

Lemma 4: The equivalence of states is transitive.

Proof. Suppose that $\theta(q, q') = true$ and $\theta(q', q'') = true$, but $\theta(q, q'') = false$. Therefore, $\exists\, s^* \in \Phi^*$ such that there is a path p associated with s^* that starts from q or q'' but not from both. Without loss of generality, let us assume that the path starts in q (a symmetrical argument can be applied if the path starts in q''). Let us consider the same sequence s^* but with respect to q' and there are two cases:

1. If there is a path starting from q' then that $\theta(q', q'') = true$ implies a contradiction and
2. If there is no path then that $\theta(q, q') = \text{true}$ also implies also a contradiction.

□

A direct consequence of lemma 4 is that $Q = B_1 \cup B_2 \cup \ldots \cup B_n$ can be obtained from the set of states such that each B_i consists of a subset of states equivalent to each other. Thus no state $q \in B_i$ and $q \in B_j$ at the same time with $i \neq j$ and therefore $B_1 \cap B_2 \cap \ldots \cap B_n = \varnothing$ (*i.e.* a partition). Once the pairs of equivalent states have been obtained (function θ) the procedure to find the state equivalent partition is as follows: For each state q a subset consisting of q and all the states that are equivalent to q, is created. Evidently, the subsets constructed in this manner form the partition of mutually equivalent states. Then, the minimal DFA can be obtained by merging all the states of each subset (*i.e.* each B_i) into one new state. The initial state of the new minimal DFA will correspond to the subset containing the initial state of the original DFA. The set T of the minimal DFA will be composed of those subsets that contain terminal states of the original DFA.

The associate FA of the SXM that corresponds to the specification can be modified and made minimal. Nevertheless, since the SXMT is a black-box technique there is no explicit description of the IUT, so in principle it is not possible to analyse the associated FA of that SXM in order to check certain properties (*i.e.* if it is minimal). However, it is well known that the minimal DFA is unique for a particular language, thus if the IUT is an SXM then it has to be a minimal DFA with the same behaviour as the associated FA of this SXM. According to [22] this result allows the application of the testing method to generate a test set that determines when the function (*i.e.* behaviour) computed by the specification and the IUT are equal, providing of course that both machines have the same type Φ.

4. SXM generalisations and variations

The success of the SXMT for DSXM has motivated the investigation of its applicability, not only when determinism is relaxed (as has been discussed in the previous section), but also to other types of EXM. In particular, the research has focused on *generalised (Eilenberg) stream X-machines* (GSXM) [8, 30, 36] and on some forms of EXM that can communicate. The models of communication for EXM are a topic that is out of the scope of the present report, and a comprehensive review of these models can be found in [46]. However, some types of EXM are related to these communication models and therefore they are studied here. Thus, apart from the GSXM, two other subclasses of EXM require special attention, the Sm-SXM and the M-SXM.

4.1 The GSXM model

A GSXM differs from a SXM in that a string of characters (including the empty string) can be added at the end of the output during the execution of each transition [8, 25, 34, 53]. Formally:

Definition 40: A GSXM is a tuple $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, I, T, m_0)$ where all the elements are as in definition 14, nevertheless:

- The type of the machine Φ is a set of non-empty processing relations:
  $\Phi: \Gamma^* \times M \times \Sigma^* \leftrightarrow \Gamma^* \times M \times \Sigma^*$

  The type is of the form:
  $\forall\, g^* \in \Gamma^*, m \in M, \phi(g^*, m, <>) = \perp$

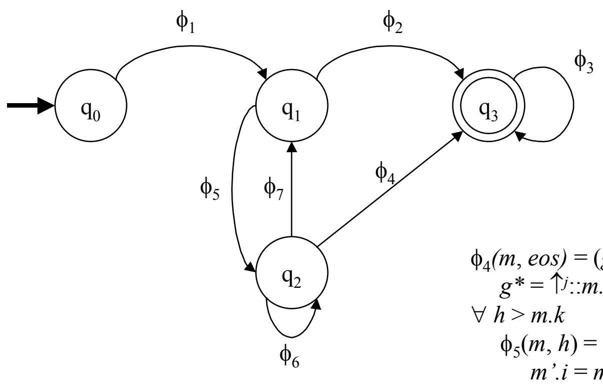
  $\forall\, g^* \in \Gamma^*, m \in M, h \in \Sigma, s^* \in \Sigma^*,$
  if $\exists\, m' \in M, t^* \in \Gamma^*$ that depends on m and h then
  $\phi(g^*, m, h::s^*) = (g^*::t^*, m', s^*)$ (*i.e.* an output sequence is concatenated at the end)
  otherwise
  $\phi(g^*, m, h::s^*) = \perp$

In other words, ρ is defined as $\rho: M \times \Sigma \rightarrow \Gamma^*$, that is a string from Γ^* is used rather than a symbol from Γ. It is easy to see that the SXM model is a subclass of the GSXM. As in the SXM, for the sake of simplicity the type of the machine will be referred to as

$\Phi: M \times \Sigma \leftrightarrow \Gamma^* \times M$ and we shall write
$\phi(m, h) = (t^*, m')$ instead of the form $\phi(g^*, m, h::s^*) = (g^*::t^*, m', s^*)$.

Convention 3: From now on the symbol ε indicates that the stream (input or output) is not modified by a function, *e.g.* $\phi(m, h) = (\varepsilon, m')$ means that a symbol h is read from the input while no symbol is written to the output.

Example 5: Now consider the problem of inserting elements in a *Binary Search Tree* (BST). The GSXM for this is presented in figure 5.



$\phi_1(m, h) = (\varepsilon, m')$ where $m'.k = h$
$\forall\ h \mid h = m.k \vee h = \uparrow \vee h = eos$
  $\phi_2(m, h) = (m.k, m)$
$\forall\ h \neq eos$
  $\phi_3(m, h) = (h, m)$
otherwise
  $\phi_3(m, h) = (\varepsilon, m)$

$\phi_4(m, eos) = (g^*, m)$ where:
  $g^* = \uparrow^j::m.k$ with $j = m.p - m.i$
$\forall\ h > m.k$
  $\phi_5(m, h) = (h, m')$ where:
    $m'.i = m.i + 1$ and $m'.p = (2 * m.i) + 1$
$\forall\ h < m.k$
  $\phi_5(m, h) = (h, m')$ where:
    $m'.i = m.i + 1$ and $m'.p = 2 * m.i$
$\forall\ h \neq eos \wedge \forall\ m.i < m.p - 1$
  $\phi_6(m, h) = (h, m)$where: $m'.i = m.i + 1$
$\forall\ m.i = m.p - 1$
  $\phi_7(m, h) = (h, m)$where: $m'.i = m.i + 1$

Figure 5

The input stream has two components, the symbol that is going to be inserted say k, followed by the BST. Given any node r in the *i-th* position of the stream, the left and the right sub trees are respectively in the (2 * i) and (2 * i + 1) positions. An available node (*i.e.* a place where a new value can be inserted) is given by the symbol ↑ also a special symbol is used to indicate the *end-of-stream* (*eos*). Additionally, it is assumed that repetitions are not allowed. Thus, to model this let us assume that the set of elements W of the BST has a relation of order, then $\Sigma = W \cup \{\uparrow, eos\}$ and $\Gamma = W \cup \{\uparrow\}$. The memory is defined as $M = (\Sigma - \{\uparrow, eos\}) \times \mathbf{N} \times \mathbf{N}$ where the first element corresponds to the symbol that is going to be inserted and two numbers are used to maintain the current node position, the left or right sub tree position. These memory components are called respectively k, i and p, and for the sake of simplicity, the same dot notation employed above is used here.

Now the intuition of our model is as follows: At the beginning, the machine reads the element that has to be inserted and stores it (ϕ_1). Note that this operation does not produce an output. After that, there are two possibilities. If the root is available or the root value is equal to k, then k is inserted in the current position (ϕ_2) and the control is transferred to state q_2 where the rest of the BST is printed (ϕ_3). The other possibility happens when the root value is greater or less than k (ϕ_5) then it is necessary to traverse the tree until the corresponding sub tree is reached (ϕ_6) and the process restart again (ϕ_7). However, if the *eos* is reached before the corresponding position of the sub tree, the output is filled with the symbol ↑ until the position before and k is written then (ϕ_4).

■

It is very natural that a number of concepts of EXM not only apply to SXM but also to GSXM like *associated* FA, *edge*, *path,* and *behaviour* (section 2.1). Additionally, the concepts of the *relation computed* (definition 15), the *output corresponding to an input sequence* (definition 19), *determinism* (definition 24), *reachability-attainability* (section 3.5) and *completely-specified* machines (definition 32) are the same for GSXM.

4.1.1 Theoretical basis for testing GSXM with output delimited type

Balanescu [8] developed a generalisation of the SXMT for GSXM that requires different (versions of) design-for-test conditions, namely *output delimited* and *strong test completeness*. Before presenting these, it is necessary to adapt the design-for-test conditions in order to fit with the definition of a GSXM as follows:

Definition 41: For any GSXM with type Φ where M' and S are as in section 3.6:

1. $\Phi \otimes (M', S)$, if
   $m_0 \in M'$ and $\forall\, \phi \in \Phi, \forall\, m \in M', \forall\, h \in S^{\phi}$, if $\phi(m, h) = (t^*, m')$ with $t^* \in \Gamma$ then $m \in M'$.

2. Φ^{C} (M', S), if
   $\Phi \otimes (M', S)$ and $\forall\, \phi \in \Phi, \forall\, m \in M', \exists\, h \in S^{\phi} \mid \phi(m, h) \neq \emptyset$.

3. Φ^{Γ} (M', S), if
   $\Phi \otimes (M', S)$ and
   $\forall\, \phi_i, \phi_j \in \Phi, \forall\, m, m'_1, m'_2 \in M', \forall\, h \in (S^{\phi i} \cap domain(\phi_j)) \cup (S^{\phi j} \cap domain(\phi_i))$ such that
   $\phi_i(m, h) = (t^*, m'_1)$ and $\phi_j(m, h) = (t^*, m'_2)$ with $t^* \in \Gamma^* \Rightarrow \phi_i = \phi_j$.

4. $\Phi^{R\Gamma}$ (M', S), if
   $\Phi \otimes (M', S)$ and
   $\forall\, \phi_i, \phi_j \in \Phi, \forall\, m, m'_1, m'_2 \in M', \forall\, h \in (S^{\phi i} \cap domain(\phi_j)) \cup (S^{\phi j} \cap domain(\phi_i))$ such that
   $\phi_i(m, h) = (t^*, m'_1)$ and $\phi_j(m, h) = (t^*, m'_2)$ with $t^* \in \Gamma^* \Rightarrow \phi_i = \phi_j$ and $m'_1 = m'_2$

5. Φ^{O} (M', S), if
   $\Phi \otimes (M', S)$ and
   $\forall\, \phi \in \Phi, \forall\, m \in M', \forall\, h \in S^{\phi} \mid \{(t^{*\prime}, m'), (t^{*\prime\prime}, m'')\} \in \phi(m, h)$, if $(t^{*\prime} = t^{*\prime\prime}) \Rightarrow (m' = m'')$.

If M' = M and $\forall\, \phi \in \Phi$, $S^{\phi} = \Sigma$ then the above is respectively denoted Φ^{C}, Φ^{Γ}, $\Phi^{R\Gamma}$ and Φ^{O}.

The property $\Phi^{R\Gamma}$ (*i.e.* output-distinguishability for relations) can be relaxed in the following form [8]:

Definition 42: if Φ is a set of relations then it is *weak-output-distinguishable* with respect to (M', S) denoted by $\Phi^{w\Gamma}$ (M', S), if

- $\Phi \otimes (M', S)$ and
- $\forall\, \phi_i, \phi_j \in \Phi, \forall\, m, m'_1, m'_2 \in M'$,
  $\exists\, h \in (S^{\phi i} \cap domain(\phi_j)) \cup (S^{\phi j} \cap domain(\phi_i)) \mid \phi_i(m, h) = (t^*, m'_1)$ and
  $\phi_j(m, h) = (t^*, m'_2)$ with $t^* \in \Gamma^* \Rightarrow \phi_i = \phi_j$ and $m'_1 = m'_2$

If M' = M and $\forall\, \phi \in \Phi$, $S^{\phi} = \Sigma$ then Φ is called *weak-output-distinguishable*, denoted by Φ^{wR}.

The proof of proposition 6 clarifies the difference between $\Phi^{R\Gamma}$ and $\Phi^{w\Gamma}$.

Proposition 6: $\Phi^{R\Gamma} \Rightarrow \Phi^{w\Gamma}$.

Proof. From definition 41.4 (based on definitions 33 and 34), $\Phi^{R\Gamma}$ establishes that the property has to be fulfilled for all the input symbols in the domain of a particular function. From definition 42, $\Phi^{w\Gamma}$ places that it is enough that the property is fulfilled for just one input symbol in the domain of the same function.

□

In addition, we have that:

Definition 43: Let Φ be $\Phi^{w\Gamma}$ then it is *strong-complete* with respect to (M', S) denoted by Φ^{CC} (M', S), if

- $\Phi \otimes (M', S)$ and
- $\forall \phi \in \Phi$, $\forall m \in M'$,
  $\forall h \in S^{\phi} \mid \phi(m, h) \neq \varnothing$ (or equivalently $(m, h) \in domain(\phi)$),

If $M' = M$ and $\forall \phi \in \Phi$, $S^{\phi} = \Sigma$ then Φ is called *strong-complete* and is denoted Φ^{CC}.

Proposition 7: $\Phi^{CC} \Rightarrow \Phi^{C}$.

Proof. From definition 41.2 (based on definitions 31) and from definition 43, following similar arguments as in proposition 6.

□

The idea of an output delimited type is that any two arbitrary paths, when processing the same input stream, will produce the same output, but only if all pair of functions in the same step of each of the paths produce the same output. Formally:

Definition 43: Let Φ be $\Phi^{w\Gamma}$ then it is as *output delimited type* denoted by Φ^{D}, if f$\phi_1, \phi_2, \ldots, \phi_n \in \Phi$ and also $\phi'_1, \phi'_2, \ldots, \phi'_n \in \Phi$, and any two output $t = t_1{::}t_2{::} \ldots {::}t_n$ and $t' = t'_1{::}t'_2{::} \ldots {::}t'_n$ with for all $t_i, t'_i \in \Gamma^*$

- $t = t'$
- $\phi_1(m_0, h_1) = (t_1, m_1)$ and $\phi'_1(m_0, h'_1) = (t'_1, m'_1)$
- $\forall\ 2 \leq i \leq n$, $\phi_i(m_{i-1}, h_i) = (t_i, m_i)$ and $\phi'_i(m'_{i-1}, h'_i) = (t'_i, m'_i)$

we have that $1 \leq i \leq n$, $t_i = t'_i$.

The definition of a GSXM with output delimiter (GSXM^{D}) presented below [30], describes a particular case of Φ^{D} in which every processing function outputs a string that always ends with a special symbol called the *output delimiter*.

Definition 44: A GSXM^{D} Λ is a GSXM where:

- The type of the machine Φ is a set of non-empty processing relations:
  Φ: $M \times \Sigma \leftrightarrow (\Gamma^*{::}\tau) \times M$ and $\tau \notin \Gamma$ (*i.e.* the output delimiter).

  The type is of the form:

  $\forall g^* \in \Gamma^*, m \in M, \phi(m, <>) = \perp$

  $\forall g^* \in \Gamma^*, m \in M, h \in \Sigma, s^* \in \Sigma^*,$
  if $\exists\ m' \in M$, $t^* \in \Gamma^*$ that depends on m and h then
  $\phi(m, h{::}s^*) = (g^*{::}t^*{::}\tau, m', s^*)$
  otherwise
  $\phi(m, h{::}s^*) = \perp$

The important part of the previous notion is, as it has already been indicated, that there is a variation of the SXMT for GSXM with Φ^{D} and Φ^{CC} [8].

4.1.2 Theoretical basis for testing non-deterministic GSXM

In [34] the basis of a method for generating test cases for another class of GSXM, namely the non-deterministic GSXM (NGSXM), is suggested. The *s*-traversal* of a path p with respect to a memory value m, is a set where each elements contains a three parts:

1. A list of input symbols from a sub-string of s^*, such that with each one it is possible to trigger the functions of a sub-path of p (including itself),

2. The final memory value computed for that sub-path,

3. The list of output produced when the sub-path is followed.

Definition 45: For a path $p = <\phi_1, \phi_2, \ldots, \phi_n>$, a memory value $m \in$ M and a input stream $s^* = h_1::h_2:: \ldots ::h_j$ where for all $h_i \in \Sigma$ and $n, j \geq 0$, if

- $n = j$ and
- $\exists\ t_1, t_2, \ldots, t_n \in \Gamma^*, m_1, m_2, \ldots, m_{n+1} \in$ M with $m = m_1$ such that $\forall\ 1 \leq i \leq n$, $\phi_i(m_i, h_i) = (t_i, m_{i+1})$ then
  $$E^{m}_{p,s^*} = \{(h_1, h_2, \ldots, h_i, m_{i+1}, t_1, t_2, \ldots, t_i) \mid 1 \leq i \leq n\}$$
  Otherwise $E^{m}_{p,s^*} = \varnothing$

Then E^{m}_{p,s^*} is called a *s*-traversal of p with respect to m*.

The notion of the above definition can be extended by using a set of paths Θ and a set of input strings S, instead of one path *p* and a single input stream *s**, as follows:

Definition 46: Let $\{E_j\}_{j\in J}$ be a family of subsets of Σ* x M x Γ* then $E^{m}_{\Theta,S} = \bigcup E_j$ is called a *S-traversal of* Θ *with respect to m* if

- $\Theta \subseteq \Phi^*$, $S \subseteq \Sigma^*$ and $m \in$ M
- $\forall\ s^* \in \Theta, \forall\ p \in \Theta, \exists\ j \in J$ such that E_j is E^{m}_{p,s^*}
- $\forall\ j \in J, \exists\ s^* \in \Theta, \exists\ p \in \Theta$, such that E_j is E^{m}_{p,s^*}

Definition 47: The set $P \subseteq \Phi^*$ is a set of *initial paths* if for all $p \in$ P, it starts from $q_0 \in$ I with initial memory value $m_0 \in$ M.

Definition 48: Let $P \subseteq \Phi^*$ the set of initial paths of a given GSXM Λ and let $S \subseteq \Sigma^*$ such that $E^{m0}_{P,S}$ (*i.e.* a S-traversal path of all the initial paths). The *S-computation* of Λ is the relation $Sc_\Lambda: \Sigma^* \leftrightarrow \Gamma^*$ given by:

$$s^*\ Sc_\Lambda\ g^* \Leftrightarrow \exists\ e \in E^{m0}_{P,S} \text{ such that } \pi_{in}(e) = s^* \text{ and } \pi_{out}(e) = g^*$$

The application of a finite number of input sequences $S \subseteq \Sigma^*$ to the IUT will produce an *S-computation* of the IUT. This is called *complete-testing assumption* [60], and without it there is no test that can ensure full fault detection for non-deterministic IUT. Moreover, for all $S \subseteq \Sigma^*$ the union of all these S-computations is equal to the relation computed by the GSXM [34], as is shown below.

Proposition 8: Let $f_\Lambda: \Sigma^* \leftrightarrow \Gamma^*$ be the relation computed by a GSXM Λ, and let H_S be the set of all the S-computations for Λ where $S \subseteq \Sigma^*$ then:

$$f_\Lambda = \bigcup_{S\subseteq\Sigma^*} \bigcup_{Sc\Lambda\in HS} Sc_\Lambda$$

Proof. From definitions 15, 47 and 48.

□

4.2 The Sm-SXM model

The Sm-SXM is another important generalisation of the SXM that allows empty-operations, following [25, 53] we have that:

Definition 49: A Sm-SXM is a tuple Λ = (Σ, Γ, M, Q, Φ, F, I, T, m_0) where all the elements are as in definition 14, nevertheless:

- The type Φ of the machine is given by $\Phi_{ne} \cup \Phi_{eo} \cup \Phi_{ei} \cup \Phi_e$:

Φ_{ne} is the set of *non-empty operations* and it is defined as,

$\forall g^* \in \Gamma^*, \forall m \in M, \phi(g^*, m, <>) = \perp$
$\forall g^* \in \Gamma^*, \forall m \in M, \forall s^* \in \Sigma^*,$
$\phi(g^*, m, s^*) = (\text{concatenate}(g^*::\rho_{ne}(m, head(s^*))), \mu_{ne}(m, head(s^*)), tail(s^*))$

Φ_{eo} is the set of *empty-output operations*, defined by,

$\forall g^* \in \Gamma^*, \forall m \in M, \phi(g^*, m, <>) = \perp$
$\forall g^* \in \Gamma^*, \forall m \in M, \forall s^* \in \Sigma^*,$
$\phi(g^*, m, s^*) = (g^*, \mu_{eo}(m, head(s^*)), tail(s^*))$

Φ_{ei} is the set of *empty-input operations*, given by,

$\forall g^* \in \Gamma^*, \forall m \in M, \forall s^* \in \Sigma^*$
$\phi(g^*, m, s^*) = (\text{concatenate}(g^*::\rho_{ei}(m)), \mu_{ei}(m), s^*)$

Φ_e is the set of *empty operations*, defined by,

$\forall g^* \in \Gamma^*, \forall m \in M, \forall s^* \in \Sigma^*,$
$\phi(g^*, m, s^*) = (g^*, \mu_e(m), s^*)$

This shows that Φ_{ne} is as in an ordinary SXM, hence any SXM is a Sm-SXM with $\Phi_{eo} = \Phi_{ei} = \Phi_e = \varnothing$. Additionally, it is easy to see from the last definition that for any Sm-SXM, its sets of operations are pair wise disjoint. It is also obvious that the empty-output-operations read an input symbol, and leave the output unchanged. On the other hand, the empty-input-operations leave unchanged the input stream while a symbol is concatenated at the end of the output. Finally, an empty-operation is a processing function ϕ of the type, which leaves unchanged both the input and the output streams. Instead of writing the functions of the type as in definition 49, we will use the shorter forms $\phi(m, h) = (t, m')$, $\phi(m, h) = (\varepsilon, m')$, $\phi(m, \varepsilon) = (t, m')$ and $\phi(m, \varepsilon) = (\varepsilon, m')$ for the functions in Φ_{ne}, Φ_{eo}, Φ_{ei} and Φ_e respectively. The Sm-SXM is also a generalisation of the GSXM model as is suggested by the following.

Proposition 9: For each GSXM there is a Sm-SXM that computes the same relation.

Proof. Let Λ be a GSXM as in definition 40, then a Sm-SXM Λ' can be obtained from it by the following procedure. For each edge in the GSXM from q to q' labelled by ϕ where $\phi(m, h) = (t^*, m')$, there are two cases depending on $w = length(t^*)$:

1. If $w = 1$ then $t^* = t \in \Gamma$ and ϕ is written as $\phi(m, h) = (t, m')$, apart from that everything remains the same in Λ' with respect to q, q' and $\phi \in \Phi_{ne}$ of Λ'.

2. If $w > 1$ then the control-states $q_1, q_2, \ldots, q_{w-1}$ are included in Λ' and ϕ is extended to be a path $p = (<q, q_1, \ldots, q_{w-1}, q'>, <\phi_1, \phi_2, \ldots, \phi_w>)$ and let us consider that $t^* = t_w:: \ldots ::t_2::t_1$ with $\forall\ 1 \le i \le w$, $t_i \in \Gamma$, therefore by defining $\phi_1(m, h) = (t_1, m') \in \Phi_{ne}$ and $\forall\ 2 \le i \le w$, $\phi_i(m', \varepsilon) = (t_i, m') \in \Phi_{ei}$, we have that $|p|(m, s^*) = \phi(m, s^*) = (g^*, m')$.

□

Example 6: From example 4, consider the function $\phi_4(m, eos) = (g^*, m)$ where $g^* = \uparrow^j::m.k$ with $j = m.p - m.i$ that goes from q_2 to q_3. By applying the procedure suggested in proposition 9, since $length(g^*) = j + 1$ we define a path $p = (<q_2, q'_1, \ldots, q'_j, q_3>, <\phi'_1, \phi'_2, \ldots, \phi'_{j+1}>)$ such that $\phi'_1(m, eos) = (\uparrow, m)$, $\forall\ 2 \le i \le j$, $\phi'_i(m, \varepsilon) = (\uparrow, m)$ and $\phi'_{j+1}(m, \varepsilon) = (m.k, m)$. Although the result of the previous proposition is important from the theoretical point of view, the fact that the length of the output is variable, makes impractical to define all the possible paths, so alternatively the memory of the machine and a loop can be constructed as follows. First an additional variable is included and let us called $m.c$ with $m_0.c = 0$. The set of states and the type of the machine are respectively extended by a new control-state q' and by three functions ϕ'_1, ϕ'_2 and ϕ'_3 where $F(q_2, \phi'_1) = q'$, $F(q', \phi'_2) = q'$ and $F(q', \phi'_3) = q_3$. Now let us define:

$\phi'_1(m, eos) = (\uparrow, m')$ where $m'.c = m.c + 1$
$\forall\ m.c \le m.p - m.i$, $\phi'_2(m, \varepsilon) = (\uparrow, m')$ where $m'.c = m.c + 1$
$\forall\ m.c > m.p - m.i$, $\phi'_3(m, \varepsilon) = (m.k, m')$

Clearly, we have a path that computes the same function as the original ϕ_4, but with the functions of the form of an Sm-SXM.

■

There are a number of immediate consequences related to the different type of EXM. In particular, the last proposition suggested the following *relation of generality* for these models: Let us say that a type of EXM_1 is more general than another type of EXM_2, denoted by $\mathrm{EXM}_1 \ \Re^{\leq} \ \mathrm{EXM}_2$ if and only if for all EXM_2 Λ there is an EXM_2 Λ' that computes the same relation. For the sake of completeness, we shall state that:

Corollary 3: SXM $\Re^{\leq}$ GSXM, GSXM $\Re^{\leq}$ Sm-SXM, Sm-SXM $\Re^{\leq}$ EXM and $\Re^{\leq}$ is transitive.

Proof. It follows from definitions 1,14, 15, 40, 49 and from proposition 9.

□

It has been shown that in any DEXM there are no trivial paths (definition 10 and proposition 1) that connect terminal states. Clearly, there is no possibility of trivial paths in either SXM or GSXM (definitions 14 and 40). This is not the case for Sm-SXM, since functions from Φ_{ei} and Φ_e leave unchanged the input stream. Having made this point, the property of determinism for Sm-SXM is expressed by the following [53]:

Definition 50: A Sm-SXM Λ is deterministic denoted by DSm-SXM, if:

- The set of initial states contains only one element q_0.
- $F: Q \times \Phi \to Q$
- $\Phi: P(X \to X)$ (*i.e.* Φ contains only functions).
- $\forall\ \phi, \phi' \in \Phi$ with $\phi \neq \phi'$, emerging from the same state then *domain*(ϕ) $\cap$ *domain*(ϕ') = $\varnothing$.

- $\forall\ q \in Q$, $m \in M$, $h \in \Sigma$, and $\forall\ \phi, \phi' \in \Phi_{ne} \cup \Phi_{eo}$, if
  (m, h) $\in$ *domain*(ϕ) and (m, h) $\in$ *domain*(ϕ') then
  either $F(q, \phi) \neq \varnothing$ or $F(q, \phi') \neq \varnothing$ (*i.e.* there is only one possible transition for any triplet q, m, h)

- $\forall\ q \in Q$, $m \in M$, $h \in \Sigma$, and $\forall\ \phi \in \Phi_{ne} \cup \Phi_{eo}$, $\phi' \in \Phi_{ei} \cup \Phi_e$, if
  (m, h) $\in$ *domain*(ϕ) and (m, ε) $\in$ *domain*(ϕ') then
  $F(q, \phi) \neq \varnothing$ or $F(q, \phi') \neq \varnothing$
  (*i.e.* there is no $q \in Q$, $m \in M$ where both a symbol h and the empty symbol ε can be taken).

- $\forall\ q \in Q$, $m \in M$, and $\forall\ \phi, \phi' \in \Phi_{ei} \cup \Phi_e$, if
  (m, ε) $\in$ *domain*(ϕ) and (m, ε) $\in$ *domain*(ϕ') then
  $F(q, \phi) \neq \varnothing$ or $F(q, \phi') \neq \varnothing$
  (*i.e.* there is no $q \in Q$, $m \in M$ where two empty transitions can be applied).

A general result of great importance is the fact that every deterministic model (DSXM, DGSXM, DSm-SXM) computes a partial function. It is worth formally recording the properties of these functions. However, we first should remark that the in the literature there have been no attempts to extend the SXMT to the Sm-SXM. Because the nature of the function from Φ_{eo},Φ_{ei} and particularly from Φ_e makes it impossible to observe the behaviour (input or output) of a given machine, which is an essential requirement of the testing method.

The Sm-SXM has been included in the present report because when the behaviour of a communicating model for SXM is treated as a single machine for testing purposes, the resulting machine could be a Sm-SXM [46, 47, 48]. Let us now examine some features related to the (partial) functions computed by DGSXMS and DSXM.

4.3 Stream functions

The GSXM model (consequently the SXM model) can guarantee that for any finite input stream the machine will stop its computation, if it is ensured that all the functions of the type are computable and terminate in a finite amount of time. From this we arrive at [53]:

Definition 51: Φ is *fully-computable*, denoted by Φ^F if $\forall\ \phi \in \Phi$, ϕ is a partial recursive function and *domain*(ϕ) is a recursive set.

This means that for each ϕ, there is an algorithm that computes it, and for all $x \in (\Gamma^* \times M \times \Sigma^*)$, x will cause the algorithm to eventually stop, in this way avoiding the halting-problem and clearly [53]:

Proposition 10: If Λ is a DGSXM with Φ^F then the relation f_Λ computed by Λ is also fully-computable.

Proof. For every input stream $s^* \in \Sigma^*$ of $length(s^*) = w$, if $k = |\Phi|$ then $f_\Lambda(s^*)$ can be determined by the application of at most k^w processing functions from Φ. If f_Λ is a function then $f_\Lambda(s^*)$ is determined by at most w consecutive partial recursive functions. In both cases f_Λ is fully-computable.

□

The function computed by the DSXM model can be either a *weak (partial) stream function* or a *(partial) stream function* [22]:

Definition 52: f: $\Sigma^* \rightarrow \Gamma^*$ is a *weak (partial) stream function*, denoted ${}^{wS}f$, if

- The function is *length preserving*:
  $\forall\, s^* \in domain(f)$, $length(s^*) = length(f(s^*))$ and

- The function is *segment preserving*:
  $\forall\, s^*, h^* \in \Sigma^*$,
  if $h^* \in domain(f)$ and $h^*::s^* \in domain(f)$ then
  there is a $g^* \in \Gamma^*$ such that $f(h^*::s^*) = f(h^*)::g^*$.

Definition 53: f: $\Sigma^* \rightarrow \Gamma^*$ is a *(partial) stream function*, denoted by ${}^{S}f$, if

- ${}^{wS}f$ and
- $\forall\, s^*, h^* \in \Sigma^*$, if $h^*::s^* \in domain(f)$ then $h^* \in domain(f)$.

An inspection of definition 52 shows that the concept of ${}^{wS}f$ is directly related to the features of a function computed by a DSXM. In brief, the first condition states that the number of symbols is the same for both the input and output sequences. The second condition fits naturally with the concatenation operator. As might be expected [22]:

Lemma 5: f: $\Sigma^* \rightarrow \Gamma^*$ is ${}^{wS}f$ if and only if there is a DSXM that computes f.

Proof. It is easily verified that every DSXM Λ computes a function that is length preserving. Because the machine is deterministic it follows that the function is segment preserving.

Now, given f a DSXM $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, I, T, m_0)$ can be constructed where:

- $M = \Sigma^*$
- $Q = \{q_1, q_2\}$
- $\Phi = \{\phi_1, \phi_2\}$ with
  ϕ_1 processes all $(s^*, s) \in M \times \Sigma$, such that $s^*::s \in domain(f)$ and
  ϕ_2 processes all $(s^*, s) \in M \times \Sigma$, such that $s^*::s \notin domain(f)$

- $F = \{(q_1, \phi_1) = q_1, (q_1, \phi_2) = q_2, (q_2, \phi_1) = q_1, (q_2, \phi_2) = q_2\}$
- $I = \{q_1\}$ if $<> \in domain(f)$, otherwise $I = \{q_2\}$
- $T = \{q_1\}$
- $m_0 = <>$

□

By now it will have become apparent that there is a strong connection between DSXM and stream functions. Naturally, this is no coincidence and the reason is to ensure that the relation works in both directions. Due to this is possible to completely characterise the class of functions computed by these machines. To close this section, let us consider DSXM whose states are all terminal $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, I, m_0)$ with $T = Q$. So far we have discussed that this machines are precisely the ones employed in the SXMT and from definition 15 it follows that:

$s^* f_\Lambda\, g^* \Leftrightarrow \exists\, q_0 \in I, q \in Q$ and p: $q_0 \rightarrow q$ such that $\pi_{out}(|p|(<>, m_0, s^*) = g^*$ and

and accordingly to [22]:

Lemma 6: f: $\Sigma^* \rightarrow \Gamma^*$ is ${}^{S}f$ if and only if there is a DSXM with all states terminal that computes f.

Proof. If f is computed by a DSXM, it follows from lemma 5 that ${}^{wS}f$. Let $h^*::s^* \in$ *domain*(f) and by making all the states of the machine terminal we have that $h^* \in$ *domain*(f) implying that f is ${}^{S}f$.

As in the previous demonstration, given f a DSXM $\Lambda = (\Sigma, \Gamma, M, Q, \Phi, F, I, m_0)$ with T = Q can be constructed where:

- $M = \Sigma^*$
- $Q = \{q_1\}$
- $\Phi = \{\phi_1\}$ with
  ϕ_1 processes all $(s^*, s) \in M \times \Sigma$, such that $s^*::s \in$ *domain*(f) and
- $F = \{(q_1, \phi_1) = q_1\}$
- $I = \{q_1\}$
- $m_0 = < >$

□

4.4 The M-SXM model

Intuitively, a M-SXM has several input and output streams, so that when the machine is operating one or more input symbols are removed and one or more output symbols are produced. Let us define formally this notion:

Definition 54: A M-SXM is a tuple $\Lambda_j^k = (\Sigma_1, \Sigma_2,\ldots, \Sigma_j, \Gamma_1, \Gamma_2,\ldots, \Gamma_k, Q, M, \Phi, F, I, T, m_0)$ where:

- The number of input streams is j and the number of output streams is k and $j, k > 0$.

- $\forall\ 1\leq i \leq j$, Σ_i is the alphabet of the *i-th* input stream and
  $\forall\ 1\leq i \leq k$, Γ_i is the alphabet of the *i-th* output stream

- The type
  $\Phi: \Gamma_1^* \times \Gamma_2^* \times\ldots\times \Gamma_k^* \times M \times \Sigma_1^* \times \Sigma_2^* \times\ldots\times \Sigma_j^* \rightarrow \Gamma_1^* \times \Gamma_2^* \times\ldots\times \Gamma_k^* \times M \times \Sigma_1^* \times \Sigma_2^* \times\ldots\times \Sigma_j^*$
  is defined as:

  $\forall\ m \in M$, $\forall\ g_1^*, g_2^*,\ldots, g_k^*$ such that $g_i^* \in \Gamma_i^*$ $1 \leq i \leq k$ and
  $\phi(g_1^*, g_2^*,\ldots, g_k^*, m, < >, < >,\ldots, < >) = \perp$

  $\forall\ m \in M$ and $\forall\ h_1, h_2,\ldots, h_j$ such that $h_i \in \Sigma_i \cup \{\varepsilon\}$ $1 \leq i \leq j$ and $(h_1, h_2,\ldots, h_j) \neq (\varepsilon, \varepsilon,\ldots, \varepsilon)$ either;
  $\phi(g_1^*, g_2^*,\ldots, g_k^*, m, h_1::s_1^*,\ldots , h_j::s_j^*) = \perp$ or

  $\exists\ m' \in M$, $t_1, t_2,\ldots, t_k$ where $t_i \in \Gamma_i \cup \{\varepsilon\}$ $1 \leq i \leq k$, and $(t_1, t_2,\ldots, t_k) \neq (\varepsilon, \varepsilon,\ldots, \varepsilon)$ such that $t_1, t_2,\ldots, t_k$ depends on m and $h_1, h_2,\ldots, h_j$,
  $\forall\ g_1^*, g_2^*,\ldots, g_k^*$ such that $g_i^* \in \Gamma_i^*$ $1 \leq i \leq k$,
  $\forall\ s_1^*, s_2^*,\ldots, s_j^*$ such that $s_i^* \in \Sigma_i^*$ $1 \leq i \leq j$,
  $\phi(g_1^*,\ldots, g_k^*, m, h_1::s_1^*,\ldots, h_j::s_j^*) = (g_1^*::t_1,\ldots, g_k^*::t_k , m', s_1^*,\ldots, s_j^*)$

- Q, M, F, I, T and m_0 are as in an ordinary SXM.

The operation of a M-SXM follows the same rules as the SXM. That is, in each control-state, given the current memory, a processing function $\phi \in \Phi$ is applied, where which of them is to be selected depends on the first symbol in some (at least one) of the input streams. Then ϕ computes a new memory-state and generates a number of output symbols (at least one), which are placed at the end of some (one or more) of the output streams. At the same time, the symbols from the input streams, which were used to select the application of ϕ, are removed. This process continues from control-state to control-state until no further computation is possible, ideally when all the input streams are empty, and the machine is in a final control-state. Then the M-SXM traverses a path while generating a sequence of outputs.

It is easy to observe that a M-SXM is not really an extension of the SXM model but rather an equivalent model that allows the specification of several (parallel) inputs and outputs. Let us prove this intuition as follows:

Lemma 7: For every SXM there is a M-SXM and for every M-SXM there is a SXM, such that both machines compute the same relation.

Proof.

Let $\Lambda = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ be a SXM then it is easy to see that
any M-SXM $\Lambda_j^k = (\Sigma_1, \Sigma_2,\ldots, \Sigma_j, \Gamma_1, \Gamma_2,\ldots, \Gamma_k, Q, M, \Phi', F', I, T, m_0)$ computes the same relationship if

$\Sigma_1 = \Sigma$, $\Gamma_1 = \Gamma$ and for any other $\Sigma_i = \{\}$ and for any other $\Gamma_i = \{\}$ and if
$\phi(g^*, m, h{::}s^*) = (g^*{::}t, m', s^*) \in \Phi$ with $F(q, \phi) = p$ then
$\phi'(g_1^*, <>,\ldots, <>, m, h_1{::}s_1^*, \varepsilon,\ldots, \varepsilon) = (g_1^*{::}t_1, <>,\ldots, <>, m', s_1^*, <>,\ldots, <>) \in \Phi'$ with $F'(q, \phi') = p$ and $g_1^* = g^*$, $h_1 = h$, $s_1 = s$, $t_1 = t$.

In the other direction, let $\Lambda_j^k = (\Sigma_1, \Sigma_2,\ldots, \Sigma_j, \Gamma_1, \Gamma_2,\ldots, \Gamma_k, Q, M, \Phi', F', I, T, m_0)$ be a M-SXM and construct a SXM $\Lambda = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ as follows:

$$\Sigma = \prod_{i=1}^{j} (\Sigma_i \cup \{\varepsilon\}) - (\varepsilon, \varepsilon,\ldots, \varepsilon)$$

$$\Gamma = \prod_{i=1}^{k} (\Gamma_i \cup \{\varepsilon\}) - (\varepsilon, \varepsilon,\ldots, \varepsilon)$$

Every $\phi'(g_1^*,\ldots, g_k^*, m, h_1{::}s_1^*,\ldots, h_j{::}s_j^*) = (g_1^*{::}t_1,\ldots, g_k^*{::}t_k, m', s_1^*,\ldots, s_j^*) \in \Phi'$ is substituted by:
$\phi(g^*, m, h{::}s^*) = (g^*{::}t, m', s^*) \in \Phi$ where:

$g^* = <g_1^*, g_2^*,\ldots, g_k^*> \in \Gamma^*$ with $\forall\ 1 \le i \le k,\ g_i^* \in \Gamma_i^*$,
$h = <h_1, h_2,\ldots, h_j> \in \Sigma$ with $\forall\ 1 \le i \le j,\ h_i \in \Sigma_i \cup \{\varepsilon\}$,
$s^* = <s_1^*, s_2^*,\ldots, s_j^*> \in \Sigma^*$ with $\forall\ 1 \le i \le j,\ s_i^* \in \Sigma_i^*$ and

$F'(q, \phi') = p$ is replaced by $F(q, \phi) = p$.

□

For the sake of simplicity, since a function of a M-SXM is completely determined by the values m' and $t_1, t_2,\ldots, t_k$, in what follows it will be referred to as:

$\Phi : M \times \Sigma_1 \times \Sigma_2 \times\ldots\times \Sigma_j \to \Gamma_1 \times \Gamma_2 \times\ldots\times \Gamma_k \times M$ with the form: $\phi(m, h_1,\ldots, h_j) = (t_1,\ldots, t_k, m')$

Seen from the viewpoint of the EXM general principles, the constraints imposed in definition 54 over the type Φ of the machine, fit naturally with the functionality of the SXM behaviour. Indeed, the conditions:

$(h_1, h_2,\ldots, h_j) \neq (\varepsilon, \varepsilon,\ldots, \varepsilon)$ and $(t_1, t_2,\ldots, t_k) \neq (\varepsilon, \varepsilon,\ldots, \varepsilon)$

constrain the machine to read at least one input symbol and to write at least one output symbol with every function $\phi \in \Phi$ performed.

Having said that, it is not hard to see that less restrictive definitions can be established with respect to the M-SXM in order to obtain models that follow the operation of a GSXM or a Sm-SXM. These models could be called the *(Eilenberg) generalised multiple-stream X-machine* (GM-SXM) or the *(Eilenberg) straight-move multiple-stream X-machine* (Sm-MSXM), and constructions similar to those in lemma 7 can be derived for them. Therefore, the use of multiple-streams is immaterial, insofar as all these pairs of models (*i.e.* SXM and M-SXM, Sm-SXM and Sm-MSXM, GSXM and GM-SXM) are equivalent, not only with respect to the relation computed by them, but also with respect to the way they operate on the streams. For these reasons and as a corollary, it is not surprising that the SXMT developed for SXM can be applied, with no modification at all, to the M-SXM, if of course it is guaranteed that design-for-test conditions are fulfilled.

However, from the specification standpoint the multiple-stream feature could be a useful mechanism for separating inputs (outputs) that belong to distinct categories, instead of having just one stream where several elements of these categories are mixed. For instance, in the specification of a cash machine the information in the magnetic tape of a card and the information typed using the keys could be considered different stream of inputs. Respect to the outputs, the information displayed on the screen, the receipts printed and the money dispensed can be abstracted in such a way that each one of them could be modelled with a distinct output stream. From this, it is apparent that the M-SXM might facilitate the specification of certain implementations, where the inputs and outputs can be (or need to be) differentiated.

Example 7: Consider the addition of n integer numbers expressed in any base b. A M-SXM can be constructed as follows: each input stream contains each one of the integer number, thus there are n input streams, where the symbols are digits. Additionally, let us assume that apart from the result that has to be an integer in base b, every time the carry computed is also written, so the machine has two output streams. Let us say that the carries and the results are respectively presented in the first and second output streams. The machine has just one control-state q_0, which clearly is the initial and final state. The alphabets and the memory are defined as $\forall\ 1 \le i \le n$, $\Sigma_i = \{0, 1, \ldots, b\text{-}1\}$ also $M = \Gamma_1 = \Gamma_2 = \{0, 1, \ldots, b\text{-}1\}$. The initial memory value is $m_0 = 0$, this value is used to keep the carry. Finally, $\Phi = \{\phi\}$ and $F(q_0, \phi) = \phi$ where:

> $\phi(m, h_1, \ldots, h_n) = (t_1, t_2, m')$ with $\forall\ s_i^* \in \Sigma_i^*\ 1 \le i \le n$, if $s_i^* = <>$ then $h_l = \varepsilon$. From definition 54, if $\forall\ s_i^* = <>$ then $\phi(m, h_1, \ldots, h_n) = \perp$ and the computation terminates.
>
> Let us say that $x = \sum h_l \mid h_l \neq \varepsilon$, thus $m' = t_1 = (x + m)\ \%\ b$, and $t_2 = (x + m)$ div b where % and div denote the modulo and integer division operators respectively.

■

5. Conclusions

The SXM model has proved to be useful for specifying and testing systems. From the theoretical point of view, the testing problem for DSXM has been solved by means of the SXMT. This method can ensure full fault coverage under certain assumptions known as the design-for-test conditions. Furthermore, some similar promising results have been achieved for other classes of EXM. Specifically, the trend in the literature points to testing when non-determinism is present. It also points to the need to extend the method to systems of communicating EXM. However, the theory of testing these systems needs further development, and more work is required on experiments with applications.

This report is a survey of relevant concepts, and results in which the theory of testing for this diversity of EXM is supported, so is intended to be a comprehensive compilation in that respect. One particular contribution is the formalisation of the M-SXM that was directly derived from the intuitive ideas of the MSS. It has been demonstrated here that a M-SXM has the same computational power as an SXM. From this, it is possible to construct a SXM from a M-SXM implying that the SXMT can be applied. Additionally, it is remarked that other classes of M-SXM can be defined to be equivalent to other classes of EXM (*e.g.* GSXM, Sm-SXM). More important is the fact that this model could provide the basis for a formal treatment of the MSS, which may allow studying what extension is possible to apply SXMT (or another testing technique) to MSS.

References

[1] S. Eilenberg. *Automata, Languages and Machines*, Vol. A, Academic press, N.Y. 1974.
[2] M. Holcombe. *X-Machines as basis for dynamic systems specification*, Software Engineering Journal, Vol. 3, No. 2, pp. 69-76, 1988.
[3] M. Holcombe. *An integrated methodology for the formal specification, verification and testing of systems*, Proc. EuroSTAR 93, London, 1993.
[4] M. Fairtlough, M. Holcombe, F. Ipate, C. Jordan, G. Laycock and Z. Duan. *Using an X-machine to model a video cassette recorder*, Current issues in Electronic modelling, Vol. 3, pp. 141-151, 1995.
[5] Z. Duan. *Modelling of Hybrid Systems*, Ph.D. Thesis, University of Sheffield, 1996.
[6] K. Bogdanov, M. Fairtlough, M. Holcombe, F. Ipate and C. Jordan, *X-machine Specification and Refinement of Digital Devices*, Research Report CS-97-16, Department of Computer Science, University of Sheffield, 1997.
[7] F. Ipate. *Using Hybrid Machines for Specifying Hybrid Software Systems*, in Proceedings of the 4th International Symposium of Economic Informatics, Bucharest, pp. 679-686, Hungry, 1999.
[8] T. Balanescu. *Generalised Stream X-Machines with Output Delimited Type*, Formal Aspects of Computing, Vol. 12, No. 6, pp. 473-484, 2000.

[9] T. Balanescu, M. Gheorghe, M. Holcombe. *A subclass of stream X-machines with underlying distributed grammars*, Proceedings, Grammars Systems 2000, Ed. R Freund & A. Kelemenova, Silesian University at Opava, Czech Republic, ISBN 80-7248-067-7, pp. 93-112, 2000.
[10] T. Balanescu, M. Gheorghe, M. Holcombe. *Deterministic Stream X-machines based on grammar systems*, in Words, sequences, grammars, languages: where computer science, linguistics and mathematics meet, Vol. I, Ed. C. Martin-Vide & Mitrana, Kluwer, 2000.
[11] G. Eleftherakis. *Model Checking and X-machine Specification*, Technical Report CS-02/00, City College, Thessaloniki, Greece, 2000.
[12] M. Gheorghe. *Generalised Stream X-Machines and Cooperating Distributed Grammar Systems*, Formal Aspects of Computing, Vol. 12, No. 6, pp. 459-472, 2000.
[13] A. Grondoudis. *X-machine Based Specification and Design for Testing of the CATV Protocol*, Ph.D. Thesis, University of Sheffield, 2000.
[14] F. Ipate and M. Popescu. *A Z type Language for Specifying X-machines*, in Proceeding of CITTI, Constanta, Romania, pp. 82-88, 2000.
[15] P. Kefalas. *Automatic translation from X-machines to Prolog*, Technical Report CS-01/00, City College, Thessaloniki, Greece, 2000.
[16] P. Kefalas. *Modelling an Agent Reactive Architecture with X-machines*, Technical Report CS-01/00, City College, Thessaloniki, Greece, 2000.
[17] P. Kefalas and A. Sotiriadou. *Transforming X-machines to Z Specifications*, Technical Report CS-06/oo, City College, Thessaloniki, Greece, 2000.
[18] P. Kefalas. *Formal Modelling of Reactive Agents as an Aggregation of Simple Behaviours*, SETN 2002, LNAI 2308, pp. 461-472, Springer-Verlag, 2002.
[19] M. Stannett and T. Simons. *Complete Behavioural Testing of Object-Oriented Systems using CCS-Augmented X-machines*. Submitted to ECCOP 2002.
[20] G. Eleftherakis and P. Kefalas. *Model Checking Safety-Critical Systems Specified as X-Machines*, to appear in Annals of Bucharest University.
[21] J. Aguado, T. Balanescu, T. Cowling, M. Gheorghe, M. Holcombe and F. Ipate. *P Systems with Replicated Rewriting and Stream X-Machines (Eilenberg Machines)*, Fundamenta Informaticae, IOS Press, No. 49, pp. 1-17, 2001.
[22] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*, Springer Verlag Series on Applied Computing, 1998.
[23] G. Laycock. *Introductions to X-machines*. Research Report CS-93-13, Department of Computer Science, University of Sheffield, 1993.
[24] G. Laycock. *The Theory and Practice of Specification-Based Software Testing*, Ph.D. Thesis, University of Sheffield, 1993.
[25] F. Ipate. *Theory of X-machines and Applications in Specification and Testing*, Ph.D. Thesis, University of Sheffield, 1995.
[26] M. Holcombe. *What are X-Machines?*, Formal Aspects of Computing, Vol 12, No. 6, pp. 418-422, 2000.
[27] F. Ipate and M. Holcombe. *An integration Testing method which is proved to find all faults*, Intern. J. Computer Math. Vol. 63, pp. 159-178, 1997.
[28] F. Ipate and M. Holcombe. *A method for refining and testing generalised machine specification*. Intern. J. Computer. Math. Vol. 68, pp. 197-219, 1998.
[29] F. Ipate and M. Holcombe. *Specification and testing using generalised machines: a presentation and a case study*, Software Testing, Verification and Reliability, Vol. 8, pp 61-81, 1998.
[30] M. Holcombe, T. Balanescu, M. Gheorghe and P. Radovici-Marculescu, *On testing generalized stream X-machines*, in Gh Paun (Ed), Recent topics in Mathematical Computational Linguistics, Romanian Academy Publishing House, pp. 130-141, 2000.
[31] R. M. Hierons and M. Harman. *Testing Conformance to a Quasi-Non-Deterministic Stream X-Machine*, Formal Aspects of Computing, Vol 12, No. 6, pp. 423-442, 2000.
[32] R. M. Hierons and M. Harman. *Testing conformance of a deterministic implementation against a non-deterministic stream X-machine*. Brunel University, October 30 2001.
[33] F. Ipate. *A method for testing non-deterministic X-machines that finds all faults*, to appear in Proc. CAIM 99, Pitesi, 1999.
[34] F. Ipate and M. Holcombe. *Generating Test Sets from Non-deterministic Stream X-Machines*. Formal Aspects of Computing, Vol. 12, No. 6, pp. 443-458, 2000.
[35] F. Ipate and M. Holcombe. *Testing non-deterministic X-machines*,in Words, sequences, grammars, languages: where computer science, linguistics and mathematics meet, Vol. II, Ed. C. Martin-Vide & Mitrana, Kluwer, 2000.
[36] F. Ipate and M. Holcombe. *Generating Test Sequences from Non-deterministic Generalised Stream X-machines*, to appear in FACS, 2001.

[37] T. Balanescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe and C. Vertan. *Communicating stream X-machines are no more than X-machines*, Journal of Universal Computer Science, Vol. 5, No. 9, pp 494-507, 1999.
[38] J. Barnard, J. Whitworth and M. Woodward. *Communicating X-machines*, Journal of Information and Software Technology, Vol. 38, pp. 401-407, 1996.
[39] J. Barnard. *COMX: a design methodology using communicating X-machines*, Inform. Software Tech., Vol. 40(5-6), pp. 271-280, 1998.
[40] J. Barnard. *Object COMX: Methodology using communicating X-machine objects*, Journal of Object-Oriented Prog. Vol. 12, No. 7, pp. 12-17, 1999.
[41] A. J. Cowling, H. Georgescu and C. Vertan. *A Structured Way to use Channels for Communication in X-machines Systems*. Formal Aspects of Computing, Vol. 12, No. 6, pp. 485-500, 2000.
[42] H. Georgescu. *Deadlock Detection in Communicating Stream X-Machine Systems*, Annals of the Bucharest University, Computer Science Series, 2000.
[43] H. Georgescu and C Vertan. *A New Approach to Communicating Stream X-machines*, Journal of Universal Computer Science, Vol. 6, No 5, pp. 490-502, 2000.
[44] P. Kefalas, G. Eleftherakis and E. Kehris. *Communicating X-Machines: A Practical Approach for Modular Specification of Large Systems*, Technical Report, CS-09-00, City College, Thessaloniki, Greece, 2000.
[45] J. Aguado and A. J. Cowling. *Design Models and the Complexity of the Testing Problem for Distributed Systems*, International Workshop on Semantic Foundations of Engineering Design Languages, SFEDL 2002, In conjunction with the 5-th European Joint Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 6-14, 2002.
[46] J. Aguado and A. J. Cowling. *Systems of Communicating X-machines for Specifying Distributed Systems*, Research Report CS-02-07, Department of Computer Science, University of Sheffield, 2002.
[47] F. Ipate and M. Holcombe. *Testing Conditions for Communicating Stream X-machine Systems*, Under consideration for publication in Formal Aspects of Computing
[48] J. Aguado. *Transfer Report*, supervised by A. J. Cowling. University of Sheffield, Nov. 2000.
[49] P. Kefalas. *X-machine Description Language: User Manual, version 1.6*, Technical Report CS-07/00, City College, Thessaloniki, Greece, 2000.
[50] P. Kefalas and E. Kapeti. *A design Language and Tool for X-machines Specifications*, in Advances in Informatics edited by D. I. Fotadis, S. D. Nikolopoulos, Word Scientific, pp. 134-145, 2000.
[51] M. Holcombe and F. Ipate. *Almost all Software Testing is Futile*. Research Report CS-95-03, Department of Computer Science, University of Sheffield, 1995.
[52] G. Brookshear. *Theory of computation: formal languages, automata and complexity*, Redwood City, Calif.; Wokingham: Benjamin/Cummings, 1989.
[53] F. Ipate and M. Holcombe. *Another look at Computability*, Informatica, Vol. 20, pp. 359-372, 1996.
[54] J. Barnard, J. Whitworth and M. Woodward. *Communicating X-machines*, Journal of Information and Software Technology, Vol. 38, pp. 401-407, 1996.
[55] T.S. Chow. *Testing Software Design Modeled by Finite-State Machines*. IEEE Transactions on Software Engineering, Vol. 4, No. 3, pp. 178-187, 1978.
[56] S. Fujiwara, G. Von Bochmann, F. khendek, M. Amalou and A. Ghedamsi. *Test Selection Based on Finite State Models*, IEEE Transactions on Software Engineering, Vol. 17, No. 6, pp. 591-603, 1991.
[57]M. P. Vasilevskii. *Failure diagnosis automata*, Kibernetika, No. 4, pp. 90-108, 1973.
[58] D. Lee and M. Yannakakis. *Principles and methods of testing finite state machines – a survey*, AT&T Bell Laboratories, Murray Hill, New Jersey. Draft Version.
[59] J. E. Hopcroft, R. Motwani and J. D. Ullman. *Introduction to automata theory, languages and computation 2nd ed.*, Boston; London, Addison-Wesley, 2000.
[60] G. Luo, G. V. Bochmann and A. Petrenko. *Test Selection Based on Communicating Non-deterministic Finite-State Machines Using a Generalised Wp-Method*, IEEE Transactions on Software Engineering, Vol. 20, No. 2, pp. 149-161, 1994.