A new model of Communicating Stream X-machine Systems

F. Ipate[1], **T. Bălănescu**[1], **P. Kefalas**[3], **M. Holcombe**[2], **G. Eleftherakis**[3]

[1]Department of Computer Science and Mathematics,
University of Pitesti, Romania
[2]Department of Computer Science,
University of Sheffield, UK
[3]Department of Computer Science,
CITY College, Thessaloniki, Greece

Abstract

One approach to formally specifying a system is to use a form of extended finite state machine called a *stream X-machine*. Several models of *communicating stream X-machines* have been devised, either for synchronous and asynchronous communication. One major problem with all these models is that they deviate from the original definition of a stream X-machine. This cause the existing stream X-machine theory or testing methods to be inapplicable for such models. This paper devises a new model of *communicating stream X-machine systems*. Unlike previous models, the communicating stream X-machines used by this model conform to the standard definition of a stream X-machine, so all theoretical results and test generation methods developed for stream X-machines will also apply to this new model.

1 Introduction

An important approach to the development of high quality software is the use of formal methods in its specification and verification. Formal specifications and models remove the possibility of ambiguity and facilitate formal, possibly automated, analysis.

One approach to formally specifying a system is to use a form of extended finite state machine called a *stream X-machine* [18], [19]. A stream X-machine (*SXM* for short) is a type of X-machine [9], [16], [18] that describes a system as a finite set of states, each with an internal store, called memory, and a number of transitions between the states. A transition is triggered by an input value, produces an output value and may alter the memory. A stream X-machine may be modelled by a finite automaton (the *associated finite automaton*) in which the arcs are labelled by function names (the *processing functions*). Thus, stream X-machines can combine the dynamic features of finite state machines with data structures, thus sharing the benefits of both these worlds. Various case studies [18], [10], [31] have demonstrated the value of the stream X-machine as a specification method, especially for interactive systems. A tool for writing stream X-machine specifications has also been constructed [27]. The *refinement* of stream X-machines [22], [25] as well as various subclasses of stream X-machines

[19], [3], [5], [12] have also been investigated. Furthermore, the minimality issue has been investigated in the context of stream X-machines [26].

Another strength of using stream X-machines to specify a system is that, under certain well defined conditions, it is possible to produce a test set that is guaranteed to determine the correctness of an implementation [20], [18], [17]. The testing method assumes that the processing functions are correctly implemented and reduces the testing of a stream X-machine to the testing of its associated finite automaton. In practice, the correctness of the processing functions is checked by a separate process: depending on the nature of a function, it can be tested using the same method or alternative functional methods [18], [21].

The method was first developed in the context of deterministic stream X-machines [20], [18] and then extended to the non-deterministic case [23]. The method in which, initially, only equivalence testing was considered, has also been extended to address conformance testing [14].

Several models of *communicating stream X-machines* have been devised [6], [2], [8], [11], [24], either for synchronous and asynchronous communication and used for modelling P-systems [1]. One major problem with all these models is that they deviate from the original definition of a stream X-machine. This cause the existing stream X-machine theory or testing methods to be inapplicable for such models. Another model, closer to the original definition, has recently been introduced and used in practical applications [29]. However, even in this case, there are differences from the standard model (i.e. each machine may use processing functions especially designed for communication), so the application of the existing theory to this model is still problematic.

This paper devises a new model of *communicating stream X-machine systems* (*CSXMS* for short). Essentially, each communicating SXM used by the model is a stream X-machine (that conforms to the original definition of the model) with an (implicit) input queue. Each communicating SXM can receive and send symbols both to other communicating SXMs or to the external environment. The interactions between the communicating SXM components of this model mimics the interactions that exist in a communicating finite state machine system [15].

There are several reasons for introducing a new model of communicating SXM. Firstly, unlike previous models, it conforms to the standard definition of a stream X-machine, so all theoretical results developed for SXMs will also apply to this new model. Secondly, it generalizes the communicating finite state machine model used to describe the control structure of specifications written in language such as Statecharts and SDL [34]. Therefore CSMSs can be used to describe both the control and the data structure of such specifications. This model of communicating SXM is thus relevant to a number of fields, including embedded systems [13] and communication protocols [36]. Finally, since the model conforms to the standard definition of a SXM, the testing method developed in the context of SXMs can be extended to this new model. This aspect is discussed in more detail in Section 6.

Before continuing, we introduce the notation used in the paper. For a finite alphabet A, A^* denotes the set of all finite sequences with members in A. ϵ denotes the empty sequence. For $a, b \in A^*$, ab denotes the concatenation of sequences a and b.

For a (partial) function $f : A \longrightarrow B$, $dom(f)$ denotes the domain of f, the subset of A for which f is defined.

For n sets $A_1, \ldots, A_n$, $\pi_i : A_1 \times \ldots \times A_n \longrightarrow A_i$ denotes the projection function, for $1 \leq i \leq n$.

2 Basic definitions

In this section the stream X-machine and other basic concepts related to it are defined. For more details see [18] or [23].

Definition 2.1 *A* stream X-Machine *(*SXM *for short) is a tuple*

$$Z = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m^0),$$

where:

- Σ *and* Γ *are finite sets called the* input alphabet *and* output alphabet *respectively;*
- Q *is the finite set of* states*;*
- M *is a (possibly) infinite set called* memory*;*
- Φ *is the* type *of* Z*, a non-empty finite set of* function symbols*. A* basic processing function
$$\phi : M \times \Sigma \longrightarrow \Gamma \times M$$
is associated with each function symbol ϕ*.*
- F *is the (partial)* next state function*,*
$$F : Q \times \Phi \longrightarrow 2^Q;$$
As for finite automata, F *is usually described by a* state-transition diagram*.*
- I *and* T *are the sets of initial and terminal states respectively,*
$$I \subseteq Q, T \subseteq Q;$$
- m^0 *is the initial memory value,*
$$m^0 \in M.$$

Thus, SXMs are X-machines for which the basic processing functions have the form $\phi : M \times \Sigma \longrightarrow \Gamma \times M$, i.e. each such function will read an input symbol, discard it and produce an output symbol while (possibly) changing the value of the memory.

It is sometimes helpful to think of an X-machine as a finite automaton with the arcs labelled by function symbols from the type Φ. The automaton $A_Z = (\Phi, Q, F, I, T)$ over the alphabet Φ is called *the associated FA* of Z. The automaton A_Z is *deterministic* if the machine has only one initial state and F maps each state/processing relation pair into at most one single state, i.e.

$$I = \{q^0\}$$

$$F : Q \times \Phi \longrightarrow Q$$

In what follows we will require that the associated automaton of the machine specification is deterministic, but this is not really a restriction of the method since [23] proves that for any arbitrary SXM there is a SXM whose associated automaton is deterministic so that the two machines compute the same relation - for the definition of the relation computed by a SXM see definition 2.4.

Note 2.1 *Throughout this paper, for $\phi_1, \phi_2 \in \Phi$, the expression $\phi_1 = \phi_2$ will denote that ϕ_1 and ϕ_2 are identical symbols (this is a stronger condition than the equality of the associated functions ϕ_1 and ϕ_2).*

Definition 2.2 *We define a configuration of a SXM by*

$$(m, q, s, g),$$

where $m \in M, q \in Q, s \in \Sigma^*, g \in \Gamma^*$. An initial configuration will have the form

$$(m^0, q^0, s, \epsilon),$$

where $q^0 \in I$ is an initial state. A final configuration will have the form

$$(m, q^f, \epsilon, g),$$

where $q^f \in T$ is a terminal state.

Definition 2.3 *A change of configuration, denoted by $\vdash$,*

$$(m, q, s, g) \vdash (m', q', s', g'),$$

is possible if $s = \sigma s'$ with $\sigma \in \Sigma$, $g' = g\gamma$ with $\gamma \in \Gamma$ and $\exists \phi \in \Phi$ such that $q' \in F(q, \phi)$ and $\phi(m, \sigma) = (\gamma, m')$. A change of configuration is called a transition *of a SXM.*

We denote by $\vdash^*$ the reflexive and transitive closure of $\vdash$.

A machine computation takes the form of a sequence of configurations that starts in an initial one and ends in a terminal one. The correspondence between the input sequence applied to the machine and the output produced gives rise to the relation computed by the machine, as defined next. In general, a SXM is non-deterministic, in the sense that the application of an input sequence can produce more than one single output sequence.

Definition 2.4 *The* relation computed by *Z,* $f_Z : \Sigma^* \longleftrightarrow \Gamma^*$, *is defined by:*

$$s f_Z g \Longleftrightarrow \exists q^0 \in I, q \in T, m \in M \text{ such that } (m^0, q^0, s, \epsilon) \vdash^* (m, q, \epsilon, g).$$

Note 2.2 *If* $s f_Z g$ *then* $length(s) = length(g)$.

Note that in certain circumstances a SXM will compute a function rather than a relation. This happens, for example, when the associated automaton is deterministic and any two processing functions associated to distinct functions symbols that emerge from the same state have disjoint domains. In this case the machine is called *deterministic.*

Definition 2.5 *An SXM Z is called* deterministic *if the following hold.*

- *The associated FA of the machine is deterministic, i.e.*
  - *Z has only one initial state, i.e.*

    $$I = \{q_0\};$$

  - *The next state function of Z maps each pair (state, processing function) onto at most one state, i.e.*

    $$F : Q \times \Phi \longrightarrow Q;$$

- *Any two distinct processing functions that label arcs emerging from the same state have disjoint domains, i.e.*

  $\forall \phi_1, \phi_2 \in \Phi$ *if* $\exists q \in Q$ *with* $(q, \phi_1), (q, \phi_2) \in dom(F)$ *then* $\phi_1 = \phi_2$ *or* $dom(\phi_1) \cap dom(\phi_2) = \emptyset$.

3 Communicating Stream X-Machines Systems (CSXMSs)

This section defines a communicating stream X-machine system and explains its behaviour.

Definition 3.1 *A* communicating stream X-machine system *(*CSXMS *for short) with n components is a tuple* $S_n = ((Z_i)_{1 \le i \le n}, E)$, *where:*

- $Z_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_i^0)$ *is the SXM with number* $i, 1 \le i \le n$.
- $E = (e_{ij})_{1 \le i,j \le n}$ *is a matrix of order* $n \times n$ *with* $e_{ij} \in \{0, 1\}$ *for* $1 \le i, j \le n$, $i \ne j$ *and* $e_{ii} = 0$ *for* $1 \le i \le n$.

A CSXMS works as follows:

- Each individual *communicating SXM* (*CSXM* for short) is a SXM plus an implicit input queue (i.e. of FIFO (first-in and first-out) structure) of infinite length; the CSXM only consumes the inputs from the queue.

- An input symbol σ received from the external environment will go to the input queue of a CSXM, say Z_j, provided that it is contained in the input alphabet of Z_j. If there exists more than one such Z_j, then $\sigma \in \Sigma_j$ will enter the input queue of one of these in a non-deterministic fashion.

- Each pair of CSXMs, say Z_i and Z_j, have two FIFO channels for communication; each channel is designed for one direction of communication. The communication channel from Z_i to Z_j is enabled if $e_{ij} = 1$ and disabled otherwise.

- An output symbol γ produced by a CSXM, say Z_i, will pass to the input queue of another CSXM, say Z_j, providing that the communication channel from Z_i to Z_j is enabled, i.e. $e_{ij} = 1$, and it is included in the input alphabet of Z_j, i.e. $\gamma \in \Sigma_j$. If these conditions are met by more than one such P_j, then γ will enter the input queue of one of these in a non-deterministic fashion. If no such Z_j exist then γ will go to the output environment.

- A CSXMS will receive from the external environment a sequence of inputs $s \in \Sigma^*$ and will send to the external environment a sequence of outputs $g \in \Gamma^*$, where $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$, $\Gamma = (\Gamma_1 \setminus In_1) \cup \ldots \cup (\Gamma_n \setminus In_n)$ with $In_i = \cup_{k \in K_i} \Sigma_k$ and $K_i = \{k \mid 1 \leq k \leq n, e_{ik} = 1\}$ for $1 \leq i \leq n$.

Example 3.1 *Consider, for example, the CSXMS $S_2 = ((Z_1, Z_2), E)$ with $e_{12} = e_{21} = 1$ and Z_1 and Z_2 as follows:*

- $\Sigma_1 = \{a, b, c\}$, $\Gamma_1 = \{x, y, z\}$, $Q_1 = \{q_1^0, q_1^1\}$, $I_1 = \{q_1^0\}$, $T_1 = Q_1$, $M_1 = \{0, 1\}$, $m_1^0 = 0$, $\Phi_1 = \{\phi_1^1, \phi_1^2, \phi_1^3\}$ *with* $\phi_1^1, \phi_1^2, \phi_1^3 : M_1 \times \Sigma_1 \longrightarrow \Gamma_1 \times M_1$ *defined by:* $\phi_1^1(m_1, a) = (x, 1 - m_1)$ *for* $m_1 \in M_1$, $\phi_1^2(1, b) = (y, 1)$, $\phi_1^3(0, c) = (z, 0)$ *and* $F_1 : Q_1 \times \Phi_1 \longrightarrow Q_1$ *defined by:* $F_1(q_1^0, \phi_1^1) = q_1^1$, $F_1(q_1^1, \phi_1^2) = q_1^0$, $F_1(q_1^1, \phi_1^3) = q_1^0$;

- $\Sigma_2 = \{u, v\}$, $\Gamma_2 = \{a, b\}$, $Q_2 = \{q_2^0, q_2^1\}$, $I_2 = \{q_2^0\}$, $T_2 = Q_2$, $M_2 = \{0, 1\}$, $m_2^0 = 0$, $\Phi_2 = \{\phi_2^1, \phi_2^2\}$ *with* $\phi_2^1, \phi_2^2 : M_2 \times \Sigma_2 \longrightarrow \Gamma_2 \times M_2$ *defined by:* $\phi_2^1(m_2, u) = (a, m_2)$, $\phi_1^2(m_2, v) = (b, m_2)$, *for* $m_2 \in M_2$ *and* $F_1 : Q_1 \times \Phi_1 \longrightarrow Q_1$ *defined by:* $F_2(q_2^0, \phi_2^1) = q_2^1$, $F_2(q_2^1, \phi_2^2) = q_2^1$.

A graphical representation of S_2 is given in Figure 1.

Suppose each CSXM is in its initial state and has initial memory value and empty input queue. If S_2 receives the input u, then it enters the input queue of Z_2. The input u will be then processed by ϕ_2^1 that outputs a and moves Z_2 in q_2^1 while its memory value remains unchanged. The value a is in the input alphabet of Z_1, so, since $e_{21} = 1$, it enters into the input queue of Z_1. a is then processed by ϕ_1^1 that outputs x and moves Z_1 in q_1^1 while the memory value becomes 1. *As x is not in the input alphabet of Z_2, it is sent to the environment as output.*

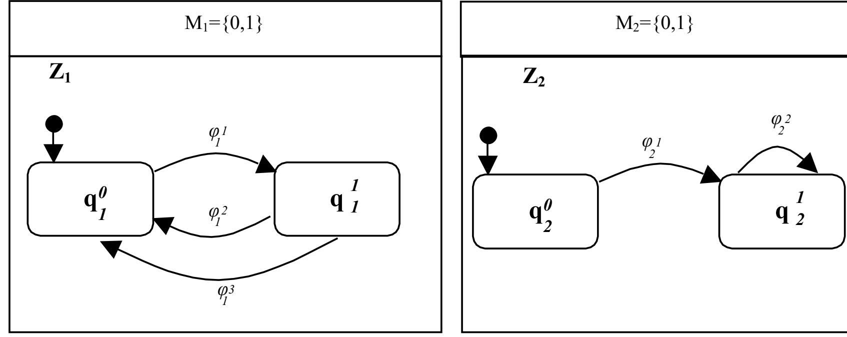

Figure 1: An abstract CSXMS with two X-machines Z_1 and Z_2

Definition 3.2 *A* configuration *of a CSXMS S_n has the form*

$$z = (z_1, \ldots, z_n, s, g),$$

where:

- $z_i = (m_i, q_i, \alpha_i), 1 \leq i \leq n$, *where* $m_i \in M_i$ *is the current value of the memory of* Z_i, $q_i \in Q_i$ *is the current state of* Z_i *and* $\alpha_i \in \Sigma_i^*$ *is the current contents of the input queue of* Z_i*;*
- s *is the current value of the input sequence;*
- g *is the current value of the output sequence.*

Definition 3.3 *An initial configuration has the form* $z^0 = (z_1^0, \ldots, z_n^0, s, \epsilon)$, *where*

$$z_i^0 = (m_i^0, q_i^0, \epsilon) \text{ with } q_i^0 \in I_i.$$

A final configuration has the form $z^f = (z_1^f, \ldots, z_n^f, \epsilon, g)$, *where*

$$z_i^f = (m_i, q_i^f, \alpha_i) \text{ with } q_i^f \in T_i.$$

Passing from a configuration z to a new configuration z' supposes that at least one of the X-machine changes its configuration, i.e. a processing function is applied.

Definition 3.4 *A change of configuration of a CSXMS S_n, denoted by* $\models$,

$$z = (z_1, \ldots, z_n, s, g) \models z' = (z'_1, \ldots, z'_n, s', g'),$$

with $z_i = (m_i, q_i, \alpha_i)$ *and* $z'_i = (m'_i, q'_i, \alpha'_i)$, *is possible if one of the following is true for some* i*:*

1. $(m'_i, q'_i, \alpha'_i) = (m_i, q_i, \alpha_i\sigma)$ *with* $\sigma \in \Sigma_i$; $z'_k = z_k$ *for* $k \neq i$; $s = \sigma s'$, $g' = g$;

2. $(m_i, q_i, \sigma\alpha_i, \epsilon) \vdash (m'_i, q'_i, \alpha'_i, \gamma)$ *with* $\sigma \in \Sigma_i$, $\gamma \in \Gamma_i \setminus In_i$; $z'_k = z_k$ *for* $k \neq i$; $s' = s$, $g' = g\gamma$;

3. $(m_i, q_i, \sigma\alpha_i, \epsilon) \vdash (m'_i, q'_i, \alpha'_i, \gamma)$ *with* $\sigma \in \Sigma_i$, $\gamma \in \Gamma_i \cap \Sigma_j$ *for some* $j \neq i$ *such that* $e_{ij} = 1$; $(m'_j, q'_j, \alpha'_j) = (m_j, q_j, \alpha_j\gamma)$; $z'_k = z_k$ *for* $k \neq i$ *and* $k \neq j$; $s' = s$, $g' = g$;

A change of configuration is called a transition *of a CSXMS.*

The three types of transitions above are called: input transitions (1), output transitions (2) and internal transitions (3). These are denoted by $\models_{inp}$, $\models_{out}$, $\models_{int}$, respectively.

For S_2 as in Example 3.1, $((0, q_1^0, \epsilon), (0, q_2^0, \epsilon), u, \epsilon) \models_{inp} ((0, q_1^0, \epsilon), (0, q_2^0, u), \epsilon, \epsilon)$ is an input transition, $((0, q_1^0, \epsilon), (0, q_2^0, u), \epsilon, \epsilon) \models_{int} ((0, q_1^0, a), (0, q_2^1, \epsilon), \epsilon, \epsilon)$ is an internal transition and $((0, q_1^0, a), (0, q_2^1, \epsilon), \epsilon, \epsilon) \models_{out} ((1, q_1^1, \epsilon), (0, q_2^1, \epsilon), \epsilon, x)$ is an output transition.

We denote by $\models^*$ the reflexive and transitive closure of $\models$.

The correspondence between the input sequence applied to the system and the output sequence produced gives rise to the relation computed by the system, as defined next.

Definition 3.5 *The* relation computed *by a CSXMS,* $f_{S_n} : \Sigma \longleftrightarrow \Gamma$ *is defined by:*

$s f_{S_n} g$ *if* $\exists z^0 = (z_1^0, \ldots, z_n^0, s, \epsilon)$ *and* $z^f = (z_1^f, \ldots, z_n^f, \epsilon, g)$ *an initial and final configurations, respectively, such that* $z^0 \models^* z^f$ *and there is no other configuration* z *such that* $z^f \models z$. *Such a sequence of transitions (i.e.* $z^0 \models^* z^f$*) will be called a* complete sequence of transitions.

Note 3.1 *The computation induced by* f_{S_n} *will always start with an input transition.*

Note 3.2 *Unlike for the relation computed by a non-communicating SXM, there may be* s *and* g *with* $length(s) \neq length(g)$ *such that* $s f_{S_n} g$. *We may even have the situation in which* $length(s) \neq 0$, $length(g) = 0$ *and* $s f_{S_n} g$, *as illustrated by the following example. If* $S_2 = ((Z_1, Z_2), E)$ *with* $e_{12} = e_{21} = 1$ *and* Z_1 *and* Z_2 *as follows:* $\Sigma_1 = \Sigma_2 = \{a, b\}$, $\Gamma_1 = \Gamma_2 = \{x\}$, $Q_1 = \{q_1^0\}$, $Q_2 = \{q_2^0\}$, $I_1 = T_1 = Q_1$, $I_2 = T_2 = Q_2$, $M_1 = M_2 = \{0\}$, $m_1^0 = m_2^0 = 0$, $\Phi_1 = \Phi_2 = \{\phi\}$ *with* ϕ *defined by* $\phi(0, a) = (x, 0)$, F_1 *defined by* $F_1(q_1^0, \phi) = q_1^0$ *and* $F_2(q_2^0, \phi) = q_2^0$ *then* $f_{S_2}(b^k) = \epsilon$, $\forall k \geq 0$.

Definition 3.6 *A CSXMS* $S_n = ((Z_i)_{1 \leq i \leq n}, E)$ *is called* deterministic *if each* Z_i *is deterministic and for each* $i \neq j$, $\Sigma_i \cap \Sigma_j = \emptyset$.

In a deterministic CSXMS, the function that processes an input symbol is always uniquely determined.

4 The product machine

We say that a CSXMS runs in a *slow environment* if inputs can be sent from the environment to the system only in situations where the input queues of all communicating SXMs are empty.

In a slow environment, if an input transition is possible then any output or internal transition is excluded and conversely. Moreover, one output transition must exist between any two input transitions, as stated in the following:

Lemma 4.1 *In a slow environment, any complete sequence of transitions is of one of the following two forms:*

1. $\models_{inp}\models_{int}^{*} ((\models_{out}\models_{inp}) \models_{int}^{*})^{*}$ *or*
2. $\models_{inp}\models_{int}^{*} ((\models_{out}\models_{inp}) \models_{int}^{*})^{*} \models_{out}$.

Proof. Follows from the explanation above. □

We say that a CSXM has a *dead-lock* if there are complete sequences of transitions of type 1 in Lemma 4.1.

Theorem 4.1 *If S_n runs in a slow environment and $sf_{S_n}g$ then ($length(g) = length(s)$ or $length(g) = length(s) - 1$). Furthermore, if S_n has no dead-lock then $length(g) = length(s)$.*

Proof. A direct consequence of Lemma 4.1. □

Theorem 4.2 *A deterministic CSXMS that runs in a slow environment computes a (partial) function rather than a relation.*

Proof. Since the CSXMS runs in a slow environment, there will be at most one non-empty queue and this will contain at most one symbol. Since the CSXMS is deterministic this symbol can be processed by at most one transition. □

We say that a CSXM has a *live-lock* if it is possible to execute an infinite number of transitions without further inputs. That is, a CSXM has a live-lock if it is possible to execute an infinite number of consecutive internal transitions. For instance, S_2 in Example 3.1 has no live-lock.

Sometimes, the live-lock may be avoided by assuming the *eventual occurrence* of some events (*fairness*). Consider the following example:

Example 4.1 *If $S_3 = ((Z_1, Z_2, Z_3), E)$ with $e_{ij} = 1$ for $i \neq j$ and Z_1, Z_2 and Z_3 as follows: $\Sigma_1 = \{a\}$, $\Sigma_2 = \Sigma_3 = \{x\}$, $\Gamma_1 = \{x\}$, $\Gamma_2 = \{a\}$, $\Gamma_3 = \{y\}$, $Q_1 = \{q_1^0\}$, $Q_2 = \{q_2^0\}$, $Q_3 = \{q_3^0\}$, $I_1 = T_1 = Q_1$, $I_2 = T_2 = Q_2$, $I_3 = T_3 = Q_3$, $M_1 = M_2 = M_3 = \{0\}$, $m_1^0 = m_2^0 = m_3^0 = 0$, $\Phi_1 = \{\phi_1\}$, $\Phi_2 = \{\phi_2\}$, $\Phi_3 = \{\phi_3\}$, with ϕ_1, ϕ_2, ϕ_3 defined by $\phi_1(0, a) = (x, 0)$, $\phi_2(0, x) = (a, 0)$, $\phi_3(0, x) = (y, 0)$, F_1, F_2, F_3 defined by $F_1(q_1^0, \phi_1) = q_1^0$ and $F_2(q_2^0, \phi_2) = q_2^0$, $F_3(q_3^0, \phi_3) = q_3^0$, then S_3 has a live-lock processing the input sequence a.*

The live-lock in Example 4.1 is due to the fact that an infinitely often enabled event (i.e. passing x to the input queue of Z_3) does not occur.

The *strong-fairness* condition guarantees the eventual occurrence of an infinitely-often enabled event. If S_3 runs under this condition then the CSXMS considered in Example 4.1 is live-lock free.

If a deterministic CSXMS runs in a slow environment and has no live-lock and dead-lock, then it can be modelled by an equivalent SXM, called the *product SXM*. This is now defined.

In the remainder of this section we will assume that the CSXMS runs in a slow environment and contains no live-lock and dead-lock.

Definition 4.1 *A* global transition *of a CSXMS S_n, denoted by $\models^g$,*

$$z \models^g z',$$

is possible if there is a sequence of transitions, $z \models z_1 \models \ldots \models z_n \models z'$ with $n \geq 1$ such that $z \models z_1$ is an input transition, $z_n \models z'$ is an output transition and for $1 \leq i \leq n-1$, $z_i \models z_{i+1}$ is an internal transition.

That is, a global transition reads an input from the external environment, writes an output to the the external environment and performs a finite number of internal transitions in between.

For S_2 as in Example 3.1, $((0, q_1^0, \epsilon), (0, q_2^0, \epsilon), u, \epsilon) \models^g ((1, q_1^1, \epsilon), (0, q_2^1, \epsilon), \epsilon, x)$ is a global transition.

Definition 4.2 *Let $S_n = ((Z_i)_{1 \leq i \leq n}, E)$ be a deterministic CSXMS with $Z_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_i^0)$, $1 \leq i \leq n$ and $E = (e_{ij})_{1 \leq i,j \leq n}$. Then a SXM $\Pi_n = (\Sigma, \Gamma, Q, M, \Phi, F_i, I_i, T_i, m^0)$ defined by:*

- $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$;
- $\Gamma = (\Gamma_1 \setminus In_1) \cup \ldots \cup (\Gamma_n \setminus In_n)$, *where* $In_i = \cup_{k \in K_i} \Sigma_k$ *and* $K_i = \{k \mid 1 \leq k \leq n, e_{ik} = 1\}$ *for* $1 \leq i \leq n$;
- $Q = Q_1 \times \ldots \times Q_n$;
- $M = M_1 \times \ldots \times M_n$;
- $\Phi = \{\phi_{qp} \mid q, p \in Q\}$, *where* $\phi_{qp} : M \times \Sigma \longrightarrow \Gamma \times M$ *is defined by:* $\phi_{qp}(m, \sigma) = (\gamma, m')$ *if* $((m_1, q_1, \epsilon), \ldots, (m_n, q_n, \epsilon), \sigma, \epsilon) \models^g ((m_1', p_1, \epsilon), \ldots, (m_n', p_n, \epsilon), \epsilon, \gamma)$, *where* $q = (q_1, \ldots, q_n)$, $p = (p_1, \ldots, p_n)$, $m = (m_1, \ldots, m_n)$, $m' = (m_1', \ldots, m_n')$;
- $F : Q \times \Phi \longrightarrow Q$ *is defined by:* $F(q, \phi_{qp}) = p$, *for* $q, p \in Q$;
- $I = I_1 \times \ldots \times I_n$;
- $T = T_1 \times \ldots \times T_n$;
- $m^0 = (m_1^0, \ldots, m_n^0)$

is called the product machine *of S_n.*

Theorem 4.3 *Let S_n be a deterministic CSXMS that runs in a slow environment and contains no live-lock or dead-lock and Π_n the product machine of S_n. Then $f_{S_n} = f_{\Pi_n}$.*

Proof. Follows from the construction of Π_n. □

5 Alternative CSXMS models

The proposed model for CSXMS conforms to the standard definition of SXM, unlike the models proposed elsewhere [6], [2], [8], [11], [24], [29]. The closest to this model is introduced in [29]. This section attempts a comparison between the two models, by presenting an example modelled in both approaches.

Definition 5.1 *In [29], a communicating X-machine model is defined as a tuple:*

$$C_n = ((\mathcal{M}_i)_{1 \leq i \leq n}, CR),$$

where

- *$\mathcal{M}_i$ is the ith SXM component that participates in the system, and*
- *CR is a communication relation between the n SXM components.*

CR is defined as a relation $CR \subseteq \mathcal{M} \times \mathcal{M}$, where $\mathcal{M} = \{\mathcal{M}_i | 1 \leq i \leq n\}$, and determines the communication channels that exist between the X-machines of the system.

A tuple $(\mathcal{M}_i, \mathcal{M}_k) \in CR$ denotes that SXM $\mathcal{M}_i$ can write a message through a communication channel to a corresponding input stream of SXM $\mathcal{M}_k$. A communicating X-machine component is therefore defined as:

$$\mathcal{M}_i = (\Sigma_{\mathcal{M}_i}, \Gamma_{\mathcal{M}_i}, Q_{\mathcal{M}_i}, M_{\mathcal{M}_i}, \Phi_{\mathcal{M}_i}, F_{\mathcal{M}_i}, q^0_{\mathcal{M}_i}, m^0_{\mathcal{M}_i}),$$

that matches the original definition of a SXM with the only difference that $\Phi_{\mathcal{M}_i}$, contains four types of functions $\phi_{\mathcal{M}_i}$:

- functions that read from standard input stream and write to standard output stream:

$$\phi_{\mathcal{M}_i}(\sigma, m) = (\gamma, m') \text{ where } \sigma \in \Sigma_{\mathcal{M}_i}, \gamma \in \Gamma_{\mathcal{M}_i}, m, m' \in M_{\mathcal{M}_i},$$

- functions that read from a communication input stream a message that is sent by another SXM $\mathcal{M}_j$ and write to standard output stream:

$$\phi_{\mathcal{M}_i}((\sigma)_{\mathcal{M}_j}, m) = (\gamma, m') \text{ where } \sigma \in \Sigma_{\mathcal{M}_i}, \gamma \in \Gamma_{\mathcal{M}_i}, m, m' \in M_{\mathcal{M}_i} \text{ and } (\mathcal{M}_j, \mathcal{M}_i) \in CR,$$

- functions that read from standard input stream and write to a communication output stream a message that is sent to another SXM $\mathcal{M}_k$:

$$\phi_{\mathcal{M}_i}(\sigma, m) = ((\gamma)_{\mathcal{M}_k}, m') \text{ where } \sigma \in \Sigma_{\mathcal{M}_i}, \gamma \in \Gamma_{\mathcal{M}_i}, m, m' \in M_{\mathcal{M}_i} \text{ and } (\mathcal{M}_i, \mathcal{M}_k) \in CR,$$

- functions that read from a communication input stream a message that is sent by another SXM $\mathcal{M}_j$ and write to a communication output stream a message that is sent to another SXM $\mathcal{M}_k$:

$$\phi_{\mathcal{M}_i}((\sigma)_{\mathcal{M}_j}, m) = ((\gamma)_{\mathcal{M}_k}, m') \text{ where } \sigma \in \Sigma_{\mathcal{M}_i}, \gamma \in \Gamma_{\mathcal{M}_i}, m, m' \in M_{\mathcal{M}_i} \text{ and } (\mathcal{M}_j, \mathcal{M}_i) \in CR \text{ and } (\mathcal{M}_i, \mathcal{M}_k) \in CR.$$

The notation $(\sigma)_{\mathcal{M}_j}$ denotes an incoming input from SXM $\mathcal{M}_i$ while $(\gamma)_{\mathcal{M}_k}$ denotes an outgoing output to SXM $\mathcal{M}_k$. Instead of communicating with only one SXM $\mathcal{M}_k$, $\mathcal{M}_i$ may communicate with all $\mathcal{M}_{k1}, \ldots, \mathcal{M}_{kp}$, sending them $\sigma_1, \ldots, \sigma_p$, respectively. In this case all $(\mathcal{M}_i, \mathcal{M}_{kj})$, $1 \leq j \leq p$, must belong to CR and $(\gamma)_{\mathcal{M}_k}$ will be replaced by $(\gamma_1)_{\mathcal{M}_{k1}} \& \ldots \& (\gamma_p)_{\mathcal{M}_{kp}}$. Apart from the standard input stream, this approach implies a set of communication input streams attached to each SXM. The total number of input streams associated with one SXM depends on the number of other SXMs, from which it receives messages.

Example 5.1 *Consider a queue of cars in front of a traffic light (figure 2) as it is adopted by [30]. The communicating system consists of two SXM, one that models the traffic light and one that models the queue of cars. In the two different approaches, the CSXMSs are:*

$$S_n = ((Light_s, CarQueue_s), [[0,1][0,0]]), \textit{ and}$$

$$C_n = ((Light_c, CarQueue_c), (Light_c, CarQueue_c))$$

In both cases, the memory of queue of cars SXM holds a sequence of cars arrived at the traffic light. The memory of the traffic light SXM holds the total number of clock ticks elapsed since the last change of colour (red to green and vice-versa) as well as the number of ticks that a colour should be displayed (duration).

While cars arrive one by one, they join the queue. A car departs from the queue only if is signaled to do so. The traffic light works under clock ticks (time units) that are used to determine the duration of each colour displayed. In the overall system, the cars should wait in the queue as long as the traffic light is red. Therefore, the communication of the two systems consists of the following: the change of light from red to green should signal the queue to allow cars to depart.

The set of states, the initial state and the next state functions for the two models are shown diagrammatically in figure 2. The rest elements of the SXMs are defined below:

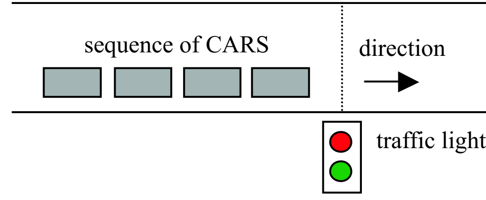

Figure 2: A sequence of cars queuing in front of a traffic light.

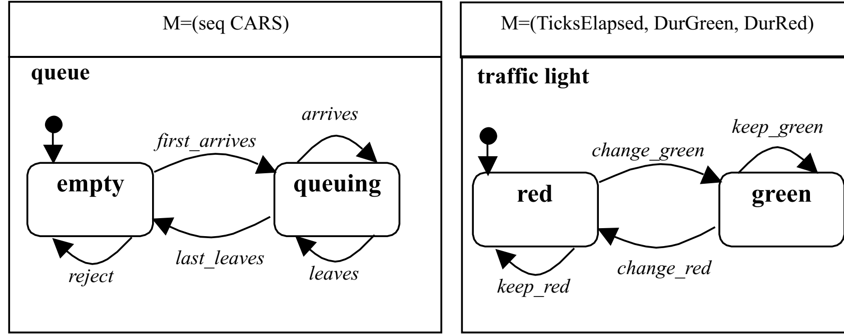

Figure 3: The set of states Q, the initial state q_0 and the next state function F of the queue and traffic light SXMs

$\Sigma_{queue} = CAR \cup \{car_leaves\}$, where $[CAR]$ is a basic type
$\Gamma_{queue} = \{FirstArrived, NextArrived, CarLeft, LastCarLeft, NoCarInQueue\}$
$M_{queue} = seq\ CAR$
$m^0_{queue} = (nil)$
$\Phi_{queue} = \{first_arrives(c, nil) = (FirstArrived, c.nil) if c \in CAR,$
$arrives(c, queue) = (NextArrived, c.queue)\ \ if\ \ c \in CAR,$
$leaves(car_leaves, c.rest) = (CarLeft, rest),$
$last_leaves(car_leaves, c.nil) = (LastCarLeft, nil),$
$reject(car_leaves, nil) = (NoCarInQueue, nil)\}$

$\Sigma_{light} = \{tick\}$
$\Gamma_{light} = \{RedColour, GreenColour\}$
$M_{light} = Ticks_elapsed \times Duration_green \times Duration_red$, where
$Ticks_elapsed, Duration_green, Duration_red \in N_0$
$m^0_{light} = (0, 30, 20)$
$\Phi_{light} =$
$\{keep_red(tick, (te, dg, dr)) = (RedColour, (te + 1, dg, dr))$ if $te + 1 < dr,$
$change_green(tick, (dr, dg, dr)) = (GreenColour, (0, dg, dr)),$
$keep_green(tick, (te, dg, dr)) = (GreenColour, (te + 1, dg, dr)$ if $te + 1 < dg,$
$change_red(tick, (dg, dg, dr)) = (RedColour, (0, dg, dr))\}$

Putting it altogether as a communicating system, the output set of light SXM should change in both approaches, S_n and C_n, to $\Gamma_{light} = \{RedColour$, $GreenColour$, $car_leaves\}$, in order to include an output signal that will trigger the departure of cars from the queue. Also, in both cases the functions $change_green$ and $keep_green$ change as follows:

- In S_n:

  $change_green(tick, (dr, dg, dr)) = (car_leaves, (0, dg, dr))$,
  $keep_green(tick, (te, dg, dr)) = (car_leaves, (te + 1, dg, dr)$ if $te + 1 < dg$,

- In C_n:

  $change_green(tick, (dr, dg, dr)) = ((car_leaves)queue, (0, dg, dr))$,
  $keep_green(tick, (te, dg, dr)) = ((car_leaves)queue, (te + 1, dg, dr)$ if $te + 1 < dg$,

- In addition, in C_n, the functions $leaves, last_leaves$ and $reject$ change to:

  $leaves((car_leaves)light, c.rest) = (CarLeft, rest)$
  $last_leaves((car_leaves)light, c.nil) = (LastCarLeft, nil)$
  $reject((car_leaves)light, nil) = (NoCarInQueue, nil))$

As far as computation is concerned the two systems behave identically. The light SXM receives a number of time units (tick) and when the appropriate time elapses it changes its state to green. While time units are still input, the light SXM outputs a signal. In S_n, the signal (car_leaves) is pick up by queue, which is the only SXM in the system that can accept such input. In C_n, the signal is directed explicitly to the queue SXM.

As shown from the formal definitions as well as the example presented above the differences between S_n and C_n are summarized in the following table:

S_n	C_n
Matrix E	Communication Relation CR
An implicit input queue to each SXM	A standard input stream to each SXM and as many communicating streams as the input communication channels attached to each SXM.
No annotation in functions	Annotations to indicate source SXM or destination SXM of input or output respectively.

The first two are conceptual differences. The matrix and the relation identify the same entity. The implicit input queue in S_n can easily be conceptualized as the product of the set of multiple implicit input streams in C_n. The third difference is subtle but a rather important one. It affects: (a) cardinality of communication, (b) modelling of large systems and (c) conformity to standard theory of X-machines.

Firstly, in C_n one can express 1 to N channels of communication between SXMs. This might be particularly useful when messages need to be broadcasted to a set of SXMs participating in the system, e.g. foraging colonies of ants or bees [28]. In the case of S_n, such communication can only be achieved by special purpose SXM which may act as multiplexers for communication and stand in between the SXMs which need to communicate in that fashion.

Secondly, modelling of large-scale systems may be less intuitive with S_n. This is because, in large systems, multiple instances of the same model may occur within the same system. For example, in a junction with more than one traffic light and consequently more queues of cars, one needs to model each instance so that each SXM receives a different input. This could be achieved by defining the input and the output set of each instance SXM in a different way. For example, one could use a tupple (Z, σ) and (Z, γ) for input and output respectively, where Z acts as an identifier for the source or the target machine. Otherwise, if this is not explicitly done, the output may be consumed by the "wrong" SXM. This is not the case with C_n, since each message "knows" its recipient.

Finally, the last difference is the one that acted as motivation for the current work. In the case of C_n, although practical development is largely facilitated by an incremental and modular process, no implication is made as to how existing strategies for testing can be incorporated to the approach. Instead, S_n provides a solid theoretical framework which, based on the conformity of CSXMS to the standard definition of SXM, can lead towards the natural extension of the testing method [18], [23]. It is feasible under certain conditions to translate some classes of C_n to an S_n and therefore to be able to apply the testing methodology in these classes as well. However, this requires further investigation.

6 Testing CSXMSs

An important strength of using stream X-machines to specify a system is that, under certain well defined conditions, it is possible to produce a test set that is guaranteed to determine the correctness of an implementation [20], [18]. We describe here the method and suggest possible ways of applying this to CSXMSs. Throughout this section we assume that SXMs and CSXMSs are deterministic.

The testing method assumes that the processing functions are correctly implemented and reduces the testing of a stream X-machine to the testing of its associated finite automaton. In practice, the correctness of the processing functions is checked by a separate process: depending on the nature of a function, it can be tested using the same method or alternative functional methods [18], [21].

Unlike the traditional extended finite state machine testing approaches [7], [32], [37], this method does not involve the construction of the equivalent finite state machine (whose states are the state/memory pairs of the original stream X-machine), and therefore does not rely on the finiteness of the memory and avoids the state explosion problem associated with this construction. Instead, in typical

applications of the method, it is successively applied to the hierarchy of stream X-machines that are created when the processing functions are considered at lower and lower levels. Thus, testing a specific function involves considering it as a computation defined by a simpler stream X-machine and so on. Ultimately, at the lowest level, the processing functions are usually quite simple and can be tested using suitable alternative methods - for example category partition testing [35] or a variant - or even assumed to be fault free if they are routines or objects from a library.

Furthermore, the method can guarantee the correctness of the implementation under test only if the processing functions meet some "design for test conditions" viz. input-completeness and output-distinguishability [20], [18], [4].

Definition 6.1 *Let* $U \subseteq \Sigma$. Φ *is called* input-complete w.r.t. U *if* $\forall \phi \in \Phi, m \in M$, $\exists \sigma \in U$ *such that* $(m, \sigma) \in dom(\phi)$. *If* $U = \Sigma$ *then* Φ *is simply called* input-complete.

This condition ensures that any processing function can be exercised from any memory value using appropriate input symbols in U.

Definition 6.2 *Let* $U \subseteq \Sigma$. Φ *is called* output-distinguishable w.r.t. U *if* $\forall \phi_1, \phi_2 \in \Phi$, $((\exists m \in M, \sigma \in U$ *with* $\pi_1(\phi_1(m, \sigma)) = \pi_1(\phi_2(m, \sigma))) \Longrightarrow \phi_1 = \phi_2)$. *If* $U = \Sigma$ *then* Φ *is simply called* output-distinguishable.

This says that we must be able to distinguish between any two different processing functions by examining outputs. If we cannot then we will not always be able to tell them apart.

For Example 3.1, Φ_1 is both input-complete and output-distinguishable, while Φ_2 is output-distinguishable but not input-complete.

The design for test conditions can be easily introduced in the definitions of the processing functions by using extra input and output symbols; a very simple algorithm is given in [18]; the extra inputs and outputs can be filtered out once the system has passed testing.

Let us see now how this method can be applied to test a CSXMS. If the CSXMS runs in a slow environment and contains no live-lock then an equivalent SXM (the product machine) can be constructed and the SXM testing method can be applied to this machine. However, this approach might not be always convenient for at least two reasons: the product machine will have to be explicitly constructed and used as a basis for test generation instead of the individual CSXMs; furthermore, this construction suffers from a combinatorial explosion (i.e. if n_i denotes the number of states of Z_i then the number of states of the product machines is of $O(\Pi(n_i)))$ and this, in turn, may produce test sets of unmanageable size.

Alternatively, the testing method can be applied to each individual CSXM; the test set for the CSXMS will be then the union of the individual test sets. However, since a CSXM Z_i cannot be tested in isolation from the rest of the systems, the following issues will have to be considered: (As above, it will be assumed that the CSXMS runs in a slow environment and contains no live-lock.)

- *feedback:* Informally, a transition t_i of a CSXM Z_i *feedbacks* in a global transition T of S_n if T contains t_i and at least one more transition from Z_i after t_i. In this case an input symbol applied to Z_i may trigger more than one transition in this CSXM, which makes the application of the method more difficult. Ideally, transitions that feedback should be avoided in a CSXM specification.

- *erroneous output masking:* Since the output symbol produced by a transition t_i of Z_i may trigger a transition in another CSXM Z_j, an erroneous output of t_i may be "masked" if there are two transitions t_j and t'_j of Z_j triggered by different inputs that have identical initial states and memory values and produce identical outputs. Indeed, let t_i be $(q_i, m_i, \sigma_i, \epsilon) \vdash (q'_i, m'_i, \epsilon, \sigma_j)$, t_j be $(q_j, m_j, \sigma_j, \epsilon) \vdash (q'_j, m'_j, \epsilon, \gamma_j)$ and t'_j be $(q_j, m_j, \sigma'_j, \epsilon) \vdash (q''_j, m''_j, \epsilon, \gamma_j)$. If t_i produces erroneous output σ'_j, it will trigger t'_j instead of t_j but this leads to the expected output. Therefore, a CSXM specification should not include transitions triggered by different inputs that have identical initial states and memory values and produce identical outputs.

- *design for test:* When design for test conditions are considered it should be taken into account the fact that not all the input symbols received by a CSXM Z_i will come from the external environment, some may come from other CSXMs. The simplest way of dealing with this situation is to enforce design for test conditions w.r.t. a set of inputs U_i that can only come from the external environment, i.e. $U_i \subseteq \Sigma_i \setminus \cup_{j \in J_i} \Gamma_j$ with $J_i = \{j \mid 1 \leq j \leq n, e_{ji} = 1\}$. Other, more complex, design for test conditions, in which the input symbols may come from other CSXMs may also be considered, but this will involve further constraints on the computation of each CSXM component.

A future paper, dedicated to CSXMS testing, will describe more formally all these aspects.

7 Conclusions

The paper introduces a new model of CSXMS. It generalizes the communicating finite state machine model used to describe the control structure of specifications written in language such as Statecharts and SDL [34]. Therefore CSXMSs can be used to describe both the control and the data structure of such specifications.

The computation of such a CSXMS is investigated, in particular for the case where the system runs in a slow environment and has no live-lock. In particular, it is shown that in these conditions the system is computationally equivalent to a SXM, called the product SXM.

The new model is compared to one of the existing CSXM models used extensively in practical applications [29]. It is shown that the two models share many general features. The model in [29] may have additional power to express

certain particular conditions but this is achieved at the expense of deviating from the standard definition of a stream X-machine.

Unlike previous models, the model of a CSXM conform to the standard definition of a stream X-machine, so all theoretical results developed for SXMs will also apply to this new model. In particular, the paper discusses the application of the existing testing method to a CSXMS.

References

[1] Aguado, J., Bălănescu, T., Cowling, T., Gheorghe, M., Holcombe, M., Ipate, F. P Systems with Replicated Rewriting and Stream X-machines (Eilenberg machines). Fundamenta Informaticae, 49 (1-3), 2002, 17-33.

[2] Bălănescu, T., Cowling, T., Georgescu, H., Gheorghe, M., Holcombe M. and Vertan, C. Communicating stream X-machines are no more than X-machines. Journal of Universal Computer Science, 5, 1999, 494-507.

[3] Bălănescu, T., Gheorghe, M. and Holcombe, M. Deterministic stream X-machines based on grammar systems. In Martin-Vide, C. and Mitrana, V. (eds), Words, sequences, grammars, languages: where biology, computer science, linguistics and mathematics meet, Volume 1. Kluwer, Dordrecht, 2000, 13-23.

[4] Bălănescu, T. Generalized stream X machines with output delimited type. Formal Aspects of Computing, 12, 2000, 473-484.

[5] Bălănescu, T., Gheorghe, M., Holcombe, M., Ipate, F. Testing collaborative agents defined as stream X-machines. Advances in Artificial Life, Proceedings of the 6th European Conference ECAL, Prague, Czech Republic, 10-14 September, Springer, Berlin, 2001, 296-305.

[6] Barnard, J., Whitworth, J., Woodward, M. Communicating X-machines. Information and Software Technology, 38, 1996, 401-407.

[7] Cheng, K.-T. and Krishnakumar, A.S. Automatic Functional Test Generation Using the Extended Finite State Machine Model. Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, ACM Press, New Orleans, 1993,86-91.

[8] Cowling, A., Georgescu, H., Vertan, C. A structured way to Use Channels for Communication in X-machine Systems. Formal Aspects of Computing, 12(6), 2000, 458-500.

[9] Eilenberg, S. Automata, languages and machines, Vol. A. Academic Press, New York, 1994

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

[11] Georgescu, H., Vertan, C. A New Approach to Communicating X-machines. Journal of Universal Computer Science, 6(5), 2000, 490-502.

[12] Gheorghe, M. Generalized Stream X-machines and Cooperating Distributed Grammar Systems. Formal Aspects of Computing, 12(6), 2001, 459-472.

[13] Heimdahl, M.P.E., Thompson, J.M., Czerny, B.J. Specification and analysis of intercomponent communication, IEEE Computer, 31, 1998, 47-54.

[14] Hierons, R.M., Harman, M. Testing conformance to a quasi-non-deterministic stream X-machine. Formal Aspects of Computing, 12(6), 2000, 423-442.

[15] Hierons, R.M. Checking States and Transitions of a set of Communicating Finite State Machines, Microprocessors and Microsystems, Special Issue on Testing and testing techniques for real-time embedded software systems, 24(9), 2001, 443-452.

[16] Holcombe, M. X-machines as a basis for dynamic system specification. Software Engineering Journal, 3, 1988, 69-76.

[17] Holcombe, M., Ipate, F. and Grondoudis, A. Complete Functional Testing of Safety-Critical Systems. Proceedinds of the 2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies, Daytona Beach, Florida, USA, 1-3 November, 199-204. Elsevier, Oxford, 1995.

[18] Holcombe, M. and Ipate, F. Correct Systems: Building a Business Process Solution. Springer Verlag, Berlin, 1998

[19] Ipate, F. and Holcombe, M. Another look at computability. Informatica, 20, 1996, 359-372.

[20] Ipate, F. and Holcombe, M. An Integration Testing Method That is Proved to Find all Faults. Intern. J. Computer Math., 63, 1997, 159-178.

[21] Ipate, F. and Holcombe, M. Specification and Testing using Generalized Machines: a Presentation and a Case Study. Software Testing, Verification and Reliability, 8, 1998, 61-81.

[22] Ipate, F. and Holcombe, M. A method for refining and testing generalized machine specifications. Intern. J. of Computer Math., 68, 1998, 197-219.

[23] Ipate, F. and Holcombe, M. Generating Test Sequences from Non-deterministic Generalized Stream X-machines. Formal Aspects of Computing, 12(6), 2000, 443-458.

[24] Ipate, F., Holcombe, M. Testing Conditions for Communicating Stream X-machine Systems. Formal Aspects of Computing, 13(6), 2002, 431-446.

[25] Ipate, F. and Holcombe, M. An Integrated Refinement and Testing Method for Stream X-Machines. Applicable Algebra in Engineering, Communication and Computing, 13(2), 2002, 67-91.

[26] Ipate, F. On the minimality of Stream X-machines, The Computer Journal, 2003 (to appear).

[27] Kefalas, P. Kapeti, E. A Design Language and Tool for X-machine Specification. Advances in Informatics. In Fotadis, D.I. and Nikolopoulos, S.D. (eds). World Scientific, 2000, 134-145.

[28] Kefalas, P., Holcombe, M., Eleftherakis, G., Gheorghe, M. A formal method for the development of agent-based systems. In: V.Plekhanova (Ed): Intelligent Agent Software Engineering, Idea Group Publishing Hershey, 2003, 68-98.

[29] Kefalas, P., Eleftherakis, G., Kehris, E. Communicationg X-machines: from theory to practice. In: Y.Manolopoulos, S.Evripidou, A.Kakas (Eds): Advances in Informatics, LNCS 2563, 2003.

[30] Kefalas, P., Eleftherakis, G., Kehris, E., Communicating X-Machines: A Practical Approach for Modular Specification of Large Systems. Information and Software Technology, 2003 (to appear).

[31] Kehris, E., Eleftherakis, G., Kefalas, P. Using X-machines to Model and Test Discrete Event Simulation Programs. In Mastorakis, N. (ed), Systems and Control: Theory and Applications. World Scientific and Engineering Society Press, Athens, 2000, 163-171.

[32] Lee, D. and Yannakakis, M. Principles and Methods of Testing Finite State Machines - A Survey. Proceedings of the IEEE, 84(8), 1996, 1090-1123.

[33] Luo, G., v. Bochmann, G., Petronko, A. Test Selection Based on Communicating Nondeterministic Finite State Machines Using a Generalized Wp-Method, IEEE Transactions on Software Engineering, 30(2), 1994, 149-161.

[34] Luo, G., Das, A., von Bochmann, G. Generating tests for control portion of SDL specifications, Protocol Test Systems vi, Elsevier, North-Holland, 1994, 51-66.

[35] Ostrand, T. J., Balcer, M. J. The Category-Partition Method for Specifying and Generating Functional Tests. Communication of the ACM, 31(6), 1989, 667-686.

[36] Tanenbaum, A.S. Computer Networks, 3rd edition, Prentice Hall, 1996.

[37] Wang, C.-J., Liu, M.T. Generating test cases for EFSM with given fault models. Proceedings of IEEE INFOCOM '93, Volume 2, 28 March - 1 April, San Francisco, CA, USA, IEEE, San Francisco, 1993, 774-781.