Testing Communicating Stream X-machines

F. Ipate[1], T. Bălănescu[1] and G. Eleftherakis[2]

[1]Department of Computer Science and Mathematics,
University of Pitesti, Romania
[2]Department of Computer Science,
CITY College, 54624 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. In practice, complex systems may be modelled by systems of communicating stream X-machines. 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. Very recently, a new model of communicating stream X-machine, that conforms to the standard definition, has been introduced. This paper investigates the applicability of the existing stream X-machine testing method to this model and identifies necessary conditions for the successful application of the method.

1 Introduction

The continuous increase in the use of computers and the fact that the required control functions increasingly demand more complex software, necessitate the need to create more reliable software. Taking into account that two thirds of the errors in software derive from incorrect informal specifications [30], the use of mathematical methods seems to form the most appropriate solution [29] in order to increase confidence in reliability of computer based systems. Any possible increase of the cost due to the use of formal methods is very small compared to the cost needed to remove software errors. In the US only, software errors cost $59.5 Billion annually to the local economy [27]!

The fact that the majority of formal languages facilitates either the data processing of the system, or the dynamics of the system but not both, and the complex notation most of them use are of the key factors for the lack of their acceptance in industry. A good practice is to use a formal language with intuitive notation that is simple for users to understand, that is capable of describing both the static and the dynamic part of a system.

A form of extended finite state machine called a *stream X-machine* [13] satisfies these requirements. A stream X-machine (*SXM* for short) is a type of X-machine [6], 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 illustrated by a finite automaton (the *associated finite automaton*) in which the arcs are labelled by function names (the *processing functions*)(figure 1). Thus, stream X-machines can combine the dynamic features of finite state machines with data structures, thus sharing the benefits of both these worlds.

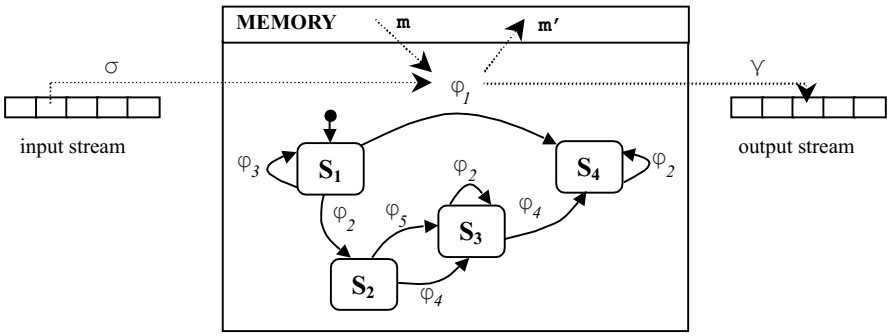

Fig. 1. Abstract X-machine model

Various case studies from different domains, like medical informatics [7], user interfaces [9], intelligent agents [23], simulation [25], biology [11], and more [13] have demonstrated the value of the stream X-machine as a specification method, especially for interactive systems.

Several issues like the *refinement* of SXM [17], the minimality issue in the context of SXM [20], model checking of SXM models [8] as well as various subclasses of stream X-machines [14, 3, 10, 2], have been investigated. A tool for writing stream X-machine specifications has also been constructed [22] based on a standard notation namely X-machine Definition Language (XMDL), used as an interchange language between developers who could share models written in XMDL for different purposes (model checker, model animator, a tool to produce the test cases etc.). 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 [13, 15]. 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 [13, 16]. The method was first developed in the context of deterministic stream X-machines [13, 15] and then extended to the non-deterministic case [18]. The method in which, initially, only equivalence testing was considered, has also been extended to address conformance testing [12].

The requirement to model distributed systems (e.g. multi-agent systems) but also the fact that no matter how simple a formal method is to learn and how powerful

and intuitive it is in modelling computer systems, it is essential that it also provides a methodology that facilitates the development of large scale and complex computer systems, created the need for communicating SXM systems.

Several models of *communicating stream X-machines* have been devised [4, 5, 19, 25] either for synchronous and asynchronous communication. They were used successfully for modelling P-systems [1, 24], multi-agent systems [23] and the behaviour of reactive agents [11]. One major problem with all these models is that they deviate, more or less, from the original definition of a stream X-machine. This cause the existing stream X-machine theory to be inapplicable for such models.

Very recently, a new model of communicating stream X-machine, that conforms to the standard definition, has been introduced [21]. Since even formally verified systems had errors found in testing phase, testing is and will always be an important part of the development cycle in improving confidence on using the final product. Thus this paper investigates the applicability of the existing stream X-machine testing method to this model and identifies necessary conditions for the successful application of the method. A simple example is used throughout this paper to demonstrate all the ideas presented.

2 Basic Definitions

In this section the stream X-machine and other basic concepts related to it are defined. For more details see [13] or [18].

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.

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 $1 \leq i \leq n$.

Definition 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 finite automaton (*FA* for short) $A_Z = (\Phi, Q, F, I, T)$ over the alphabet Φ is called *the associated FA* of Z.

Definition 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 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 $\vdash$.

Definition 4. *An SXM Z is called* deterministic *if the following hold.*

– *The associated FA A_Z 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 5. *A* communicating stream X-machine system*(*CSXMS *for short) with n components is a tuple $S_n = ((Z_i)_{1\leq i\leq n}, E)$, 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* $i, 1 \leq i \leq n$.
- $E = (e_{ij})_{1\leq i,j\leq n}$ *is a matrix of order* $n \times n$ *with* $e_{ij} \in \{0,1\}$ *for* $1 \leq i,j \leq n$, $i \neq j$ *and* $e_{ii} = 0$ *for* $1 \leq i \leq n$.

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 1. As an example, the model of a secure door that opens by entering a 3-digit code as a CSXMS is presented. Assuming that the valid code is 492, the definition of this model follows.

Consider the CSXMS $secure\_door = ((digicode, door), E)$ with $e_{digicode\ door} = e_{door\ digicode} = 1$ and $digicode$ and $door$ as follows:

- $\Sigma_{digicode} = \{ok, door - closes\} \cup DIGITS$, where $DIGITS = \{0, \ldots, 9\}$,
  $\Gamma_{digicode} = \{0, \ldots, 9, open, initialise, ignore - digit, reject - input\}$,
  $Q_{digicode} = \{ready, accepting, code\ entered\}$,
  $I_{digicode} = \{ready\}$, $T_{digicode} = Q_{digicode}$,
  $M_{digicode} = (seq\ DIGITS, seq\ DIGITS)$, $m^0_{digicode} = (nil, < 492 >)$,
  $\Phi_{digicode} = \{reject, input\_digit, ignore, finish, lock\}$ with $reject, ignore, input\_digit$, $finish, lock : M_{digicode} \times \Sigma_{digicode} \longrightarrow \Gamma_{digicode} \times M_{digicode}$ defined by:

- $reject((cs, code), ok) = (reject - input, (nil, code))$ if $cs \neq code$,
- $input\_digit((cs, code), d) = (d, (cs^{\wedge}d, code))$ if $\#cs < \#code \wedge d \in DIGITS$,
- $ignore((cs, code), d) = (ignore{-}digit, (cs, code))$ if $\#cs = \#code \wedge d \in DIGITS$,
- $finish((cs, code), ok) = (open, (cs, code))$ if $cs = code$,
- $lock((cs, code), door - closes) = (initialise, (nil, code))$

and $F_{digicode} : Q_{digicode} \times \Phi_{digicode} \longrightarrow Q_{digicode}$ defined as illustrated in figure 2.

– $\Sigma_{door} = \{open, close\}$,
$\Gamma_{door} = \{door - opens, door - closes, open - ignored, close - ignored\}$,
$Q_{door} = \{closed, opened\}$,
$I_{door} = \{closed\}$, $T_{door} = Q_{door}$,
M_{door} is the set of positive integers; M_{door} is used to count the time the door was opened,
$m^0_{door} = 0$,
$\Phi_{door} = \{open\_door, close\_door, ignore\_open, ignore\_close\}$ with $open\_door$, $close\_door$, $ignore\_open$, $ignore\_close : M_{door} \times \Sigma_{door} \longrightarrow \Gamma_{door} \times M_{door}$ defined by:
  - $open\_door(n, open) = (door - opens, n + 1)$,
  - $close\_door(n, close) = (door - closes, n)$,
  - $ignore\_open(n, open) = (open - ignored, n)$,
  - $ignore\_close(n, close) = (close - ignored, n)$,

and $F_{door} : Q_{door} \times \Phi_{door} \longrightarrow Q_{door}$ defined as illustrated in figure 2.

A graphical representation of *secure_door* is given in Figure 2.

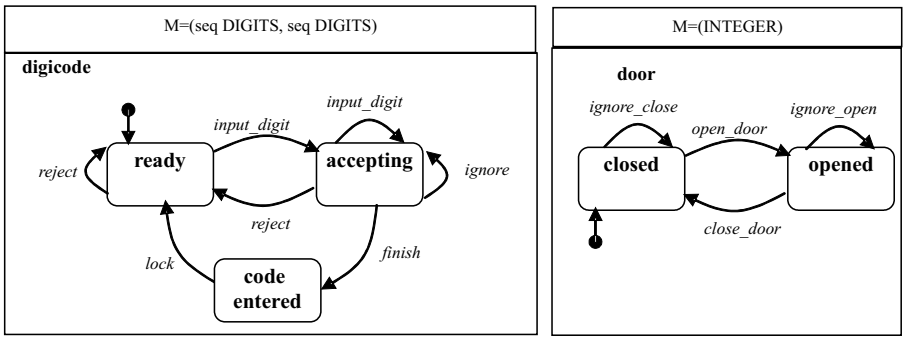

Fig. 2. The CSXMS *secure_door* with two X-machines *digicode* and *door* modelling a secure door that opens by entering a 3-digit password using a keypad

To demonstrate a possible computation of the above described CSXMS, suppose each CSXM is in its initial state and has initial memory value and empty input queue. If *secure_door* receives the input 4, then it enters the input queue of *digicode*. The input 4 will be then processed by *input_digit* that outputs 4, which is not in the input alphabet of *door* and it is sent to the environment as output, and moves *digicode* in *accepting* while its memory value is changed to $(< 4 >, < 492 >)$. Accordingly if

secure_door receives the inputs $9, 2$ they will both enter the input queue of *digicode* and will be both processed, one after the other, by *input_digit* that outputs 9 and 2 respectively, which are not in the input alphabet of *door* and they are sent to the environment as output, and moves *digicode* in *accepting* again while its memory value is changed to $(< 492 >, < 492 >)$. If then *secure_door* receives the input *ok*, it also enters the input queue of *digicode* and is processed by $finish$, which moves *digicode* to *code entered* state and outputs *open*. The value *open* is in the input alphabet of *door* so, since $e_{digicode\ door} = 1$, it enters into the input queue of *door* and is then processed by *open_door*, that outputs $door - opens$ and changes the memory of *door* to (1). Now the value $door - opens$ is not in the input alphabet of *digicode* and is sent to the environment as output. Finally if the following input *secure_door* received is *close*, then it enters the input queue of *door*, it is processed by *close_door* which leaves the memory unchanged, moves *door* to the state *closed* and it outputs $door - closes$. This value belongs to the input alphabet of the *digicode* so, since $e_{door\ digicode} = 1$, it enters into the input queue of *digicode* and is processed by *lock*, which moves *digicode* to *ready* state and outputs *initialise*. As *initialise* is not in the input alphabet of *door*, it is sent to the environment as output.

Definition 6. *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 7. *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 8. *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 *secure_door* as in Example 1,
$(((nil, < 492 >), ready, \epsilon), (0, closed, \epsilon), 4, \epsilon) \models_{inp}$
$(((nil, < 492 >), ready, 4), (0, closed, \epsilon), \epsilon, \epsilon)$ is an input transition,
$(((< 492 >, < 492 >), accepting, ok), (0, closed, \epsilon), \epsilon, \epsilon) \models_{int}$
$(((< 492 >, < 492 >), code\ entered, \epsilon), (0, closed, open), \epsilon, \epsilon)$ is an internal transition, and
$(((< 49 >, < 492 >), accepting, 2), (0, closed, \epsilon), \epsilon, \epsilon) \models_{out}$
$(((< 492 >, < 492 >), accepting, \epsilon), (0, closed, \epsilon), \epsilon, 2)$ is an output transition.

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

Definition 9. *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.

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.

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, *secure_door* in Example 1 has no live-lock.

4 Testing SXMSs

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 [13, 15]. 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 [13, 16].

Unlike the traditional extended finite state machine testing approaches [26], 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 [28] 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 [13, 15].

Definition 10. *Let* $In = \{In_\phi \mid \phi \in \Phi\}$ *with* $In_\phi \subseteq \Sigma$ *for* $\phi \in \Phi$. Φ *is called* input-complete w.r.t. In *if* $\forall \phi \in \Phi, m \in M, \exists \sigma \in In_\phi$ *such that* $(m, \sigma) \in dom(\phi)$. *If* $In_\phi = \Sigma$ *for* $\phi \in \Phi$ *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 In.

Definition 11. *Let* $In = \{In_\phi \mid \phi \in \Phi\}$ *with* $In_\phi \subseteq \Sigma$ *for* $\phi \in \Phi$. Φ *is called* output-distinguishable w.r.t. In *if* $\forall \phi_1, \phi_2 \in \Phi$, $((\exists m \in M, \sigma \in (In_{\phi_1} \cup In_{\phi_2})$ *with* $\pi_1(\phi_1(m, \sigma)) = \pi_1(\phi_2(m, \sigma))) \Longrightarrow \phi_1 = \phi_2)$. *If* $In_\phi = \Sigma$ *for* $\phi \in \Phi$ *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 1, Φ_{door} is both input-complete and output-distinguishable, while $\Phi_{digicode}$ is output-distinguishable but not input-complete (although it can easily be with the addition of one function that will accept $door - closes$ as input, just to ignore it, and two transitions from *ready* to itself and from *accepting* to itself also, and one function to ignore any digit input and the *ok* input with one transition from *code entered* to itself).

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 [13]; the extra inputs and outputs can be filtered out once the system has passed testing.

5 Testing CSXMSs

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) [21] 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.)

5.1 Feedback

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. Clearly, transitions that feedback reduce the effectiveness of the testing method and should be avoided in a CSXM specification.

5.2 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. sets of inputs that can only come from the external environment, i.e.

$\phi \in \Phi_i \Longrightarrow In_\phi \subseteq \Sigma_i \setminus \cup_{j \in J_i} \Gamma_j$, where $J_i = \{j \mid 1 \leq j \leq n, e_{ji} = 1\}$.

5.3 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". There are several situations in which this may happen:

- the output produced by t_i triggers no other transition while the erroneous output triggers a transition t_j in Z_j that produces the same output as t_i. Indeed, let t_i be $(q_i, m_i, \sigma_i, \epsilon) \vdash (q_i', m_i', \epsilon, \gamma_i)$ and t_j be $(q_j, m_j, \sigma_j, \epsilon) \vdash (q_j', m_j', \epsilon, \gamma_i)$. If t_i produces erroneous output σ_j, it may trigger t_j and this leads to the expected output.
- there are two transitions t_j and t_j' of Z_j triggered by different inputs coming from Z_i 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.
- there are two transitions, t_j of Z_j and t_k of Z_k triggered by different inputs coming from Z_i that 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_k be $(q_k, m_k, \sigma_k, \epsilon) \vdash$

$(q''_k, m''_k, \epsilon, \gamma_j)$. If t_i produces erroneous output σ_k, it will trigger t_k instead of t_j but this leads to the expected output.

Erroneous output masking can be avoiding by enforcing additional conditions on the CSXM specifications.

The simplest way is to outlaw all transitions that may be triggered by outputs produced by other CSXMs when the test inputs are applied. That is, if

$$Out_i = \cup_{\phi \in \Phi_i} \pi_1(\phi(M_i \times In_i))$$

and

$$L = \cup_{1 \leq i \neq i \leq n} (Out_i \cap \Sigma_j)$$

we will require that $L = \emptyset$.

Alternatively, erroneous output masking is avoided if the following two conditions are met:

Definition 12. *S_n is called* output-separable *if* $\forall 1 \leq i, j \leq n$, $((\exists \phi_i \in \Phi_i, \phi_j \in \Phi_j, \sigma_i \in \Sigma_i, \sigma_j \in \Sigma_j, m_i \in M_i, m_j \in M_j$ *with* $\pi_1(\phi_1(m_i, \sigma_i)) = \pi_1(\phi_1(m_j, \sigma_j))) \Longrightarrow i = j)$.

This says that any two distinct CSXMs cannot produce identical outputs irrespective of the states, inputs and memory values. This prevents the situation where an erroneous output is masked by a different CSXMs than the one that processes the correct output.

In this case, the only possibility of erroneous output occurs when the correct output and the erroneous output are both processed by the same CSXM. The following condition disallows this possibility, too:

Definition 13. *S_n is called* input-observable *if* $\forall 1 \leq i \leq n$, $\sigma, \sigma' \in Sigma_i$, $((\exists \phi, \phi' \in \Phi_i, m \in M_i, q \in Q_i$ *with* $(q, \phi), (q, \phi') \in dom(F_i)$ *and* $\pi_1(\phi(m, \sigma)) = \pi_1(\phi'(m, \sigma'))) \Longrightarrow \sigma = \sigma')$.

This says that no CSXM may have transitions triggered by different inputs that have identical initial states and memory values and produce identical outputs.

If the conditions stated in Definition 12 and Definition 13 are met then erroneous output masking will not be possible in a CSXMS.

5.4 CSXMs with a Reset Operation

The input-observability condition can be relaxed if all CSXMs have a reset operation. A CSXM Z_i is said to have a reset operation if for any state q_i and memory value m_i of Z_i there is an input σ_i that forces Z_i in its initial state q_i^0 and memory value m_i^0 from q_i and m_i.

In this case, the condition above can be replaced by a less restrictive variant:

Definition 14. *S_n is called* initially input-observable *if* $\forall 1 \leq i \leq n$, $\sigma, \sigma' \in Sigma_i$, $((\exists \phi, \phi' \in \Phi_i, m \in M_i$ *with* $(q_i^0, \phi), (q_i^0, \phi') \in dom(F_i)$ *and* $\pi_1(\phi(m, \sigma)) = \pi_1(\phi'(m, \sigma')))$ $\Longrightarrow \sigma = \sigma')$.

If all CSXMs have a reset operation, then each time an input symbol is applied to a CSXM Zi, the other CSXMs can be brought to their initial states and memory values by the reset operation, so it is enough to be able to identify the inputs that trigger transitions that emerge from these initial states and memory values.

6 Conclusions

Rather than applying the SXM testing method to the equivalent SXM of a CSXMS - an approach that may very often prove impractical due to the size of the resulting SXM - this paper investigates the application of the method to individual CSXM components. It identifies additional problems relating to the communicating nature of these machines and identifies conditions in which these problems are avoided. It also shows how these conditions can be relaxed for a particular class of CSXMs, i.e. CSXMs with a reset operation. Further work will concern the extension of these results to the non-deterministic case.

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), 17-33 (2002)
2. Bălănescu T., Gheorghe M., 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,* Vol.1. Kluwer, (2000), 13-23
3. Bălănescu T., Gheorghe M., Holcombe M., Ipate F. Testing Collaborative Agents Defined as Stream X-machines. *Proceedings 6th European Conference on Advances in Artificial Life (ECAL)*, Prague, Czech Republic, (2001), 296-305
4. Barnard J., Whitworth J., Woodward M. Communicating X-machines. *Information and Software Technology,* 38: 401-407 (1996)
5. Cowling A., Georgescu H., Vertan C. A Structured Way to Use Channels for Communication in X-machine Systems. *Formal Aspects of Computing,* 12(6):458-500 (2000)
6. Eilenberg S. *Automata, Languages and Machines,* Vol.A. Academic Press, New York, (1994)
7. Eleftherakis G. A Formal Framework for Modelling and Validating Medical Systems. *Proceedings MEDINFO Conference*, Vol.1, London, UK, (2001) 13-17
8. Eleftherakis G., Kefalas P., Sotiriadou A. XmCTL: Extending Temporal Logic to Facilitate Formal Verification of X-machine Models. *Analele Universitatii Bucuresti, Matematica-Informatica*, 50:79-95 (2001)
9. Fairtlough M., Holcombe M., Ipate F., Jordan C. Laycock G., Duan Z. Using an X-machine to Model a Video Cassette Recorder. *Current issues in Electronic Modeling,* 3:141-161 (1995)
10. Gheorghe M. Generalized Stream X-machines and Cooperating Distributed Grammar Systems. *Formal Aspects of Computing,* 12(6):459-472 (2001)
11. Gheorghe M., Holcombe M., Kefalas P., Computational Models of Collective Foraging, *Proceedings 4th International Workshop on Information Processing in Cells and Tissues (IPCAT)*, Luven, Belgium, (2001)

12. Hierons R.M., Harman, M. Testing Conformance to a Quasi-non-deterministic Stream X-machine. *Formal Aspects of Computing,* 12(6):423-442 (2000)
13. Holcombe M., Ipate F. *Correct Systems: Building a Business Process Solution.* Springer, Berlin, (1998)
14. Ipate F., Holcombe M. Another Look at Computability. *Informatica*, 20:359-372 (1996)
15. Ipate F., Holcombe M. An Integration Testing Method That is Proved to Find all Faults. *International Journal Computer Mathematics*, 63:159-178 (1997)
16. Ipate F., Holcombe M. Specification and Testing Using Generalized Machines: a Presentation and a Case Study. *Software Testing, Verification and Reliability,* 8:61-81 (1998)
17. Ipate F., Holcombe M. A Method for Refining and Testing Generalized Machine Specifications. *International Journal of Computer Mathematics*, 68:197-219 (1998)
18. Ipate F., Holcombe M. Generating Test Sequences from Non-deterministic Generalized Stream X-machines. *Formal Aspects of Computing,* 12(6):443-458 (2000)
19. Ipate F., Holcombe M. Testing Conditions for Communicating Stream X-machine Systems. *Formal Aspects of Computing,* 13(6):431-446 (2002)
20. Ipate F. On the Minimality of Stream X-machines, *The Computer Journal,* 2003 (to appear)
21. Ipate F., Bălănescu T., Kefalas P., Holcombe M., Eleftherakis, G. A New Model of Communicating Stream X-machine Systems, *Romania Journal of Information Science and Technology,* Romanian Academy, 2003 (to appear)
22. Kefalas P., Kapeti E. A Design Language and Tool for X-machine Specification. In *Advances in Informatics*, Fotadis D.I., Nikolopoulos S.D. (eds). World Scientific, (2000), 134-145
23. Kefalas P., Holcombe M., Eleftherakis G., Gheorghe M. A Formal Method for the Development of Agent-based sSystems. In: *Intelligent Agent Software Engineering,* V.Plekhanova (ed), Idea, Hershey, (2003), 68-98
24. Kefalas P., Eleftherakis G., Holcombe M. and Gheorghe M. Simulation and Verification of P Systems using Communicating X-machines. *BioSystems*, (2003)
25. Kefalas P., Eleftherakis G., Kehris E. Communicationg X-machines: from Theory to Practice. In: Y.Manolopoulos, S.Evripidou, A.Kakas (eds): *Advances in Informatics, Springer LNCS Vol.2563*, (2003)
26. Lee D., Yannakakis M. Principles and Methods of Testing Finite State Machines - a Survey. *Proceedings of the IEEE,* 84(8):1090-1123 (1996)
27. NIST (US National Institute of Standards and Technology), Software Errors Cost U.S. Economy $59.5 Billion Annually, *http://www.nist.gov/public_affairs/releases/n02-10.htm,* (28 June 2002)
28. Ostrand T.J., Balcer M.J. The Category-Partition Method for Specifying and Generating Functional Tests. *Communication of the ACM,* 31(6):667-686 (1989)
29. Saiedian H. et al., An Invitation to Formal Methods, *IEEE Computer*, 29(4):16-32 (1996)
30. Villemeur A., *Reliability, Availability, Maintainability and Safety Assessment*, John Wiley, (1991)