Formal Aspects of Computing (2002) 13: 431–446
© 2002 BCS

Formal Aspects of Computing

Testing Conditions for Communicating Stream X-machine Systems

Florentin Ipate[1] and Mike Holcombe[2]

[1]Faculty of Science, University of Pitesti, Pitesti, Romania
[2]Department of Computer Science, University of Sheffield, Sheffield, UK

Abstract. X-machines were proposed by Holcombe as a possible specification language and since then a number of further investigations have demonstrated that the model is intuitive and easy to use. In particular, *stream X-machines* (*SXM*), a particular class of X-machines, have been found to be extremely useful in practice. Furthermore, a method of testing systems specified as SXMs exists and is proved to detect all faults of the implementation provided that the system meets certain *"design for test conditions"*. Recently, a *system of communicating SXMs* was introduced as a means of modelling parallel processing. This paper proves that each communicating machine component can be transformed in a straightforward manner so that the entire system will behave like a single stream X-machine - the equivalent SXM of the system. The paper goes on to investigate the applicability of the SXM testing method to a system of communicating SXMs and identifies a class of communicating SXMs for which the equivalent SXM of the system meets the "design for test conditions".

Keywords: Communicating (extended) fsm; Communicating (stream) X-machines; Design for test-conditions; Testing

1. Introduction

Introduced by Eilenberg [Eil74] in 1974, *X-machines* are proposed by Holcombe [Hol88] as a basis for a possible specification language and since then a number of further investigations have demonstrated that the model is intuitive and easy to use as well as general enough to cater for a wide range of applications [HoI98], [FHI95].

In its essence an X-machine is like a finite state machine, but with one important difference. A basic data set, X, is identified together with a set of basic processing relations or functions, Φ, which operate on X. Each arrow in the finite state machine diagram is then labelled by a relation or function from Φ, the sequences of state transitions in the machine determine the processing of the data set and thus the relation or function computed. The data set X can contain information about the internal memory of a system as well as different sorts of output behaviour so it is possible to model very general systems in a transparent way.

Correspondence and offprint requests to: Mike Holcombe, Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK. e-mail: m.holcombe@dcs.shef.ac.uk

A number of important classes of X-machines have been identified and studied [Ipa95], [HoI98]. Typically, the classes are defined by restrictions on the underlying data set X and the set of basic processing relations or functions, Φ, of the machines. In particular, *stream X-machines* (*SXM*) have been found to be extremely useful in practice and most of the theory developed so far has concentrated on this particular class of machines. As suggested by their name, SXMs are those in which the input and the output sets are streams of symbols. The machine processes the input stream and produces, in turn, a stream of outputs and a regularly updated internal memory. Each processing relation or function processes one memory/ input pair to produce one or more output/ memory pairs. Thus, SXMs are generalisations of finite state machines (FSM) [LeY96], similar to extended finite state machines (EFSM) [LeY96], [ChK93], [IpH96], [IpH98a]: here, the variables are replaced by a memory and the sets of predicates and assignments are replaced by a set of processing functions or relations.

Furthermore, a testing theory based on the SXM model has been developed, which is a generalisation of the existing FSM testing theory [Cho78], [FBK91], [LBP94]. A method of testing systems specified as SXM exists and is proved to detect *all faults* of the implementation provided that two major requirements are met. Firstly, the system has to satisfy some *"design for test conditions"*. Basically these conditions require that the specified system has to have detectable behaviour under all conditions. Without such conditions, the system may be very difficult, or indeed impossible, to test. Secondly, the method assumes that the implementations of the basic processing functions Φ are correct, thus the system implementation uses the same set of basic processing functions as the specification. The testing method is first developed in the context of deterministic SXMs [IpH97] and then its application is generalised to non-deterministic (generalised) SXMs [IpH01]. In what follows, this latter variant will be considered and it will referred to as the *SXM testing method*.

Recently, Balanescu et al. [BCG99] specified a model for a *communicating stream X-machine* (*CSXM*) as an extension of the SXM model. Basically, a CSXM is a SXM that can communicate with others via input and output ports and a communication matrix. In this paper, we investigate the applicability of the SXM testing method to a system of CSXMs. We prove that, by performing a slight transformation on each communicating machine, the system itself will behave like a single SXM. We also investigate to what extent the resulting SXM will meet the "design for test conditions" required by the SXM testing method. We show that these conditions can be slightly relaxed without affecting the validity of the SXM testing method and we identify a class of CSXMs for which the resulting SXM will meet these *"relaxed design for test conditions"*.

Before we go any further, we introduce the notations used in this paper. When considering sequences of inputs or outputs we will use A^* to denote the set of finite sequences with members in A. ϵ will denote the empty sequence. For a sequence $a \in A^*$, $length(a)$ will denote the number of elements of a ($length(\epsilon) = 0$). For $a, b \in A^*$, ab will denote the concatenation of sequences a and b. For a relation $f : A \longleftrightarrow B$,

$$dom(f) = \{a \in A \mid f(a) \neq \emptyset\}.$$

For $b \in B$,

$$f^{-1}(b) = \{a \in A \mid afb\}.$$

2. Basic definitions

This section introduces briefly the stream X-machine model and basic concepts and notation associated with it. For more details see [HoI98] or [IpH01].

Definition 2.1. A *stream X-Machine* (*SXM*) P is a tuple

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

as follows:

- 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 set of *processing relations* of P, a set of basic non-empty relations (partial functions) that the machine can use, of the form

$$\phi : M \times \Sigma \longleftrightarrow \Gamma \times M.$$

- F is a (partial) function,

$$F : Q \times \Phi \longrightarrow 2^Q.$$

F is called the *next-state function* of P and is usually described as 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*,

$$m^0 \in M.$$

Thus, SXM are X-machines for which the basic processing relations (partial functions) are of the form $\phi : M \times \Sigma \longleftrightarrow \Gamma \times M$, i.e. each such relation will read an input symbol, discard it and produce an output symbol.

It is sometimes helpful to think of an X-machine as an automaton with the arcs labelled by relations from the set Φ. The automaton $A = (\Phi, Q, F, I, T)$ over the alphabet Φ is called the *associated automaton* of P. The automaton A is called *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.

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

$$F : Q \times \Sigma \longrightarrow Q$$

The application of the SXM method will require that the associated automaton of the machine specification is deterministic, but this is not really a restriction of the method since [IpH01] 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.

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 $(m, \sigma)\phi(\gamma, m')$. 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 state and ends in a terminal state. 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 a SXM*, $f : \Sigma^* \longleftrightarrow \Gamma^*$, is defined by:

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

Note that in certain circumstances a SXM will compute a function rather than a relation. This happens when the associated automaton is deterministic, Φ is a set of (partial) functions (rather than relations) and any two distinct processing functions that emerge from the same state have disjointed domains. In this case the machine is called *deterministic*. In this paper, however, we will consider the, more general, *non-deterministic* case and the application of the SXM testing method to this case.

Given a state of the machine, q, all the memory values that can be computed by the machine in this state are said to be attainable in q.

Definition 2.5. Given a state $q \in Q$, a memory value $m \in M$ is called *attainable in* q if there exist $q^0 \in I, s \in \Sigma^*, g \in \Gamma^*$ such that $(m_0, q^0, s, \epsilon) \vdash^* (m, q, \epsilon, g)$. The set of all memory values attainable in q will be denoted by $Mattain_q(P)$, i.e.

$$Mattain_q(P) = \{m \in M \mid \exists q^0 \in I, s \in \Sigma^*, g \in \Gamma^* \text{ such that } (m_0, q^0, s, \epsilon) \vdash^* (m, q, \epsilon, g)\}.$$

Given a processing relation ϕ, a memory value $m \in M$ is said to be ϕ-attainable if m can be attained in at least one state from which ϕ emerges.

Definition 2.6. Given a processing relation $\phi \in \Phi$, a memory value $m \in M$ is called *ϕ-attainable* if there exists $q \in Q$ such that $F(q, \phi) \neq \emptyset$ and $m \in Mattain_q(P)$. The set of all ϕ-attainable memory values will be denoted by $Mattain_\phi(P)$, i.e.

$$Mattain_\phi(P) = \bigcup_{q \in Q, F(q,\phi) \neq \emptyset} Mattain_q(P).$$

3. Design for testing conditions

This section defines the *"design for test conditions"* of the SXM testing method and shows that these can be slightly relaxed without affecting the validity of the method.

There are two design for test conditions (completeness and output-distinguishability) that the machine has to meet and in general these can be defined with respect to any pair of sets U, V that meet the following closure property:

Definition 3.1. Let $V \subseteq M$ and $U = \{U^\phi \mid \phi \in \Phi\}$ a family of non-empty subsets of $\Sigma, U^\phi \subseteq \Sigma$, indexed by Φ. Then Φ is called *closed w.r.t.* (V, U) if the following are true:

- $m^0 \in V$
- $\forall \phi \in \Phi, m \in V, \sigma \in U^\phi, \gamma \in \Gamma$, if $(m, \sigma)\phi(\gamma, m')$ then $m' \in V$.

That is, we identify a set of memory values V, with $m_0 \in V$, and, for each processing relation or function ϕ, a set of input symbols U^ϕ such that ϕ will produce only memory values in V when acting on memory values in V and inputs in U^ϕ. In the worst case, we can choose $V = M$ and $U^\phi = \Sigma, \phi \in \Phi$.

Definition 3.2. Let $V \subseteq M$ and $U = \{U^\phi \mid \phi \in \Phi\}$ with $U^\phi \subseteq \Sigma$, such that Φ is closed w.r.t. (V, U). Then Φ is called *complete w.r.t.* (V, U) if:

$$\forall \phi \in \Phi, m \in M, \exists \sigma \in U^\phi \text{ such that } \phi(m, \sigma) \neq \emptyset.$$

If $V = M$ and $\forall \phi \in \Phi U^\phi = \Sigma$ then Φ is called *complete.*

In other words, any processing relation ϕ will be able to process all memory values of V using inputs in U^ϕ. This guarantees that any path of the associated automaton can be exercised from the initial state and initial memory value.

Definition 3.3. Let $V \subseteq M$ and $U = \{U^\phi \mid \phi \in \Phi\}$ with $U^\phi \subseteq \Sigma$, such that Φ is closed w.r.t. (V, U). Then Φ is called *output-distinguishable w.r.t.* (V, U) if:

$$\forall \phi_1, \phi_2 \in \Phi, m, m'_1, m'_2 \in V, \sigma \in (U_{\phi_1} \cap dom(\phi_2)) \cup (U_{\phi_2} \cap dom(\phi_1)), \gamma \in \Gamma$$

$$(m, \sigma)\phi_1(\gamma, m'_1) \text{ and } (m, \sigma)\phi_2(\gamma, m'_2) \Longrightarrow \phi_1 = \phi_2 \text{ and } m'_1 = m'_2.$$

If $V = M$ and $\forall \phi \in \Phi U^\phi = \Sigma$ then Φ is called *output-distinguishable.*

What this is saying is that the memory/input pair processed and the output produced will uniquely determine the processing relation applied and the next memory value. Note that if Φ is a set of partial functions rather than relations then the output-distinguishability condition has the following, simpler, form:

$$\forall \phi_1, \phi_2 \in \Phi, m, m'_1, m'_2 \in V, \sigma \in (U_{\phi_1} \cap dom(\phi_2)) \cup (U_{\phi_2} \cap dom(\phi_1)), \gamma \in \Gamma$$

$$\phi_1(m, \sigma) = (\gamma, m'_1) \text{ and } \phi_2(m, \sigma) = (\gamma, m'_2) \Longrightarrow \phi_1 = \phi_2.$$

These two design for test conditions are required of the specification machine if this is to be tested successfully using the SXM method [IpH01]. They will ensure that the specification under testing will have detectable behaviour, that is it will be possible to exercise any path in the machine from the initial memory using appropriate input sequences and each output sequence produced will identify uniquely the path followed. However, these two conditions can be slightly relaxed without affecting the validity of the SXM method. This is shown next.

Definition 3.4. Let $V \subseteq M$ and $U = \{U^{\phi} \mid \phi \in \Phi\}, U^{\phi} \subseteq \Sigma$, such that Φ is closed w.r.t. (V, U). Then P is called *input-complete w.r.t.* (V, U) if:

$$\forall \phi \in \Phi, m \in Mattain_{\phi}(P) \cap V, \exists \sigma \in U^{\phi} \text{ such that } \phi(m, \sigma) \neq \emptyset.$$

If $V = M$ and $\forall \phi \in \Phi U^{\phi} = \Sigma$ then P is called *input-complete.*

Definition 3.5. Let $V \subseteq M$ and $U = \{U^{\phi} \mid \phi \in \Phi\}, U^{\phi} \subseteq \Sigma$, such that Φ is closed w.r.t. (V, U). Then P is called *output-distinguishable w.r.t.* (V, U) if:

$$\forall \phi_1, \phi_2 \in \Phi, m \in Mattain_{\phi_1}(P) \cap Mattain_{\phi_2}(P), m'_1, m'_2 \in V, \sigma \in (U_{\phi_1} \cap dom(\phi_2)) \cup (U_{\phi_2} \cap dom(\phi_1)), \gamma \in \Gamma$$

$$(m, \sigma)\phi_1(\gamma, m'_1) \text{ and } (m, \sigma)\phi_2(\gamma, m'_2) \Longrightarrow \phi_1 = \phi_2 \text{ and } m'_1 = m'_2.$$

If $V = M$ and $\forall \phi \in \Phi U^{\phi} = \Sigma$ then P is called *output-distinguishable.*

If Φ is a set of partial functions rather than relations then the above condition has the following, simpler, form:

$$\forall \phi_1, \phi_2 \in \Phi, m \in Mattain_{\phi_1}(P) \cap Mattain_{\phi_2}(P), m'_1, m'_2 \in V, \sigma \in (U_{\phi_1} \cap dom(\phi_2)) \cup (U_{\phi_2} \cap dom(\phi_1)), \gamma \in \Gamma$$

$$\phi_1(m, \sigma) = (\gamma, m'_1) \text{ and } \phi_2(m, \sigma) = (\gamma, m'_2) \Longrightarrow \phi_1 = \phi_2.$$

It is easy to see that the above two requirements are particular cases of the two "design for test conditions" of definitions 3.2 and 3.3, where only memory values that are ϕ-attainable are considered. These two requirements will be referred to as *"relaxed design for test conditions"*. However, the SXM testing method remains valid if the "design for test conditions" are replaced by the "relaxed design for test conditions"- the proof of theorem 5.1 in [IpH01], which is the theoretical basis for this method, is not affected by this replacement.

4. Communicating Stream X-Machines (CSXM)

This section defines a communicating stream X-machine and explains the behaviour of a communicating stream X-machine system. The definitions in this section are broadly from [BCG99].

Definition 4.1. A *communicating stream X-Machine* (*CSXM*) is a tuple

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

as follows:

- Σ and Γ are finite sets called the *input alphabet* and the *output alphabet* respectively.
- Q is a finite set of states; $Q = Q' \cup Q''$ with $Q' \cap Q'' = \emptyset$, where Q' is the set of *ordinary states* and Q'' the set of *communication states.*
- M is a (possibly) infinite set called the *memory.*
- Φ is the set of processing relations of P;

  $$\Phi = \Phi' \cup \Phi'' \text{ with } \Phi' \cap \Phi'' = \emptyset,$$

  where Φ' is the set of *ordinary relations* of the machine and Φ'' is the set of *communication functions* of the machine. The elements of Φ' are relations (partial functions) of the form

  $$\phi' : IN \times M \times OUT \times \Sigma \longleftrightarrow \Gamma \times IN \times M \times OUT.$$

  The elements of Φ'' are partial functions whose form will be given later.

- F is the "next state function", a (partial) function

$$F : Q \times \Phi \longrightarrow 2^Q \text{ with } dom(F) \subseteq (Q' \times \Phi') \cup (Q'' \times \Phi'').$$

That is, any arc that emerges from an ordinary state is labelled by an ordinary relation whereas any arc that emerges from a communication state is labelled by a communication function.

- $I \subseteq Q$ is the set of *initial states* and $T \subseteq Q$ is the set of *terminal states.*
- $m^0 \in M$ is the *initial memory.*
- IN and OUT are two sets called the *input port* and the *output port* respectively; they take values from the memory M or have the undefined value λ, i.e.

$$IN, OUT \subseteq M \cup \{\lambda\}, \text{ with } \lambda \notin M.$$

Similar to the case of a standard SXM, the automaton $A = (\Phi, Q, F, I, T)$ over the alphabet Φ is called the associated automaton of P.

Definition 4.2. A *communicating stream X-machine system*(*CSXMS*) with n components is a tuple $S_n = ((P_i)_{1 \leqslant i \leqslant n}, C)$, where:

- $P_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_i^0, IN_i, OUT_i)$ is the CSXM with number $i, 1 \leqslant i \leqslant n$.
- C is a set of matrices of order $n \times n$ used for communicating between the machines. For $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n$, we will denote

$$C_{ij} = \{c_{ij} \mid \ c \in C, \text{ where } c = (c_{ij})_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n}\}.$$

The machines of the system communicate as follows.

- For $c \in C$ and each pair (i, j), $i \neq j$, $c_{ij} \in C_{ij}$ is used as a temporary buffer for passing messages from the output port of the X-machine P_i to the input port of the X-machine P_j, thus $OUT_i \subseteq C_{ij} \subseteq IN_j$ for $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n$, $i \neq j$. The messages are memory values passed through the input and output ports of the machines. The symbol λ is used to describe an empty buffer. There is no need for an X-machine to pass a message to itself so we will denote $C_{ii} = \{\theta\}$ for $1 \leqslant i \leqslant n$. The symbol θ will be used only for this purpose, i.e. $\theta \notin C_{ij}$ for $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n$, $i \neq j$.
- Initially the communication buffers are all empty, i.e. $c_{ij}^0 = \lambda$ for $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, i \neq j$, where $c^0 = (c_{ij}^0)_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n}$ is the initial communication matrix.
- The X-machine Pi can only read from the i-th column and write to the i-th row of the communication matrix. These operations are performed by the communication functions of P_i. These are partial functions $\phi_i'' : IN_i \times OUT_i \times C \longrightarrow IN_i \times OUT_i \times C$ as follows.
  - For $1 \leqslant i \leqslant n$, we denote by MVO_i the set of the moves from the output port of the i-th machine to a matrix location c_{ij}. These are called *output moves.* Moving a value from the output port out_i to a matrix location c_{ij} is possible only when c_{ij} is empty and out_i is not empty. Thus

$$MVO_i = \{mvo_{ij} \mid 1 \leqslant j \leqslant n, j \neq i\},$$

where

$$mvo_{ij} : IN_i \times OUT_i \times C \longrightarrow IN_i \times OUT_i \times C$$

is a partial function defined by:

$$mvo_{ij}(in_i, out_i, c) = (in_i, \lambda, c'), \text{ if } out_i \neq \lambda \text{ and } c_{ij} = \lambda,$$

where

$$c'_{ij} = out_i \text{ and } c'_{uv} = c_{uv} \text{ for } 1 \leqslant u \leqslant n, 1 \leqslant v \leqslant n, (u, v) \neq (i, j).$$

  - For $1 \leqslant i \leqslant n$, we denote by MVI_i the set of the moves from a matrix location c_{ij} to the input port of the i-th machine. These are called *input moves.* Moving a value from a matrix location c_{ji} to the input port in_i is possible only when in_i is empty and c_{ji} is not empty. Thus

$$MVI_i = \{mvi_{ji} \mid 1 \leqslant i \leqslant n, j \neq i\},$$

where

$$mvi_{ji} : IN_i \times OUT_i \times C \longrightarrow IN_i \times OUT_i \times C$$

is a partial function defined by:

$$mvi_{ji}(\lambda, out_i, c) = (c_{ji}, out_i, c'), \text{ if } c_{ji} \neq \lambda,$$

where

$$c'_{ji} = \lambda \text{ and } c'_{uv} = c_{uv} \text{ for } 1 \leqslant u \leqslant n, 1 \leqslant v \leqslant n, (u,v) \neq (j,i).$$

Thus the set of the communication functions of Pi, Φ''_i, is made up only of input and output moves, i.e.

$$\Phi'' \subseteq MVO_i \cup MVI_i, 1 \leqslant i \leqslant n.$$

Note that P_i can execute mvo_{ij} only when c_{ij} is empty whereas P_j can execute mvi_{ij} only when c_{ij} is not empty, thus the operations concerning the same position (i, j) of the current matrix c are done under mutual exclusion.

Also note that the ordinary relations will leave the communication matrix unchanged while the communication functions will either read from the i-th column or write to the i-th row of the communication matrix. Thus the rest of the matrix will remain unaffected by the application of any processing relation or partial function.

For an uniform treatment, all the relations (partial functions) of $\phi_i \in \Phi_i$ can be extended to

$$\overline{\phi_i} : (IN_i \times M_i \times OUT_i \times C) \times (\Sigma_i \cap \{\epsilon\}) \longleftrightarrow (\Gamma_i \cap \{\epsilon\}) \times (IN_i \times M_i \times OUT_i \times C).$$

The extended relations (partial functions) will operate only on the variables for which ϕ_i is defined and leave the rest unchanged. That is, if

$$\phi_i : IN_i \times M_i \times OUT_i \times \Sigma_i \longleftrightarrow \Gamma_i \times IN_i \times M_i \times OUT_i$$

is an ordinary relation then

$$((in_i, m_i, out_i, c), \sigma_i)\overline{\phi_i}(\gamma_i, (in'_i, m'_i, out'_i, c)) \text{ if } (in_i, m_i, out_i, \sigma_i)\phi_i(\gamma_i, in'_i, m'_i, out'_i).$$

If

$$\phi_i : IN_i \times OUT_i \times C \longrightarrow IN_i \times OUT_i \times C$$

is a communication function then

$$\overline{\phi_i}((in_i, m_i, out_i, c), \epsilon) = (\epsilon, (in'_i, m_i, out'_i, c')) \text{ if } \phi_i(in_i, out_i, c) = (in'_i, out'_i, c').$$

For simplicity, in what follows we will use ϕ_i in place of $\overline{\phi_i}$; that is, the same notation will be used for a processing relation or its extension and they will be distinguished by their domains.

Example 4.1. We model a system composed of two agents that can read the letter x and produce one of the letters y and z so that the number of the y's produced by the first agent is less than or equal to the number of y's produced by the second agent. The two agents are specified by two CSXMs, P_1 and P_2. From time to time P_2 sends to P_1 the number of y's produced since the last transmission. The internal memory of P_1 records the difference between the number of y's produced by P_2 till the last transmission and the number of y's produced by P_1 up to the current moment. The internal memory of M_2 records the number of y's produced by M_2 since the last transmission.

$$\Sigma_1 = \Sigma_2 = \{x\},$$

$$\Gamma_1 = \Gamma_2 = \{y, z\},$$

$M_1 = M_2$ are both the set of positive integers,

$$m_1^0 = m_2^0 = 0,$$

P_2 will send messages to P_1 but not vice-versa, so

$$IN_1 = OUT_2 = C_{21} = M_2 \cup \{\lambda\},$$

$$IN_2 = OUT_1 = C_{12} = \{\lambda\},$$

The transition diagrams of the two machines may be found in figure 1, the initial states are A and D, respectively. A, B and D are ordinary states whereas C and E are communication states. In state C, P_1 will read from c_{21}; in state E, P_2 will write to c_{21}. The ordinary relations are:

$$\Phi'_1 = \{f, g\}, \Phi'_2 = \{h\},$$

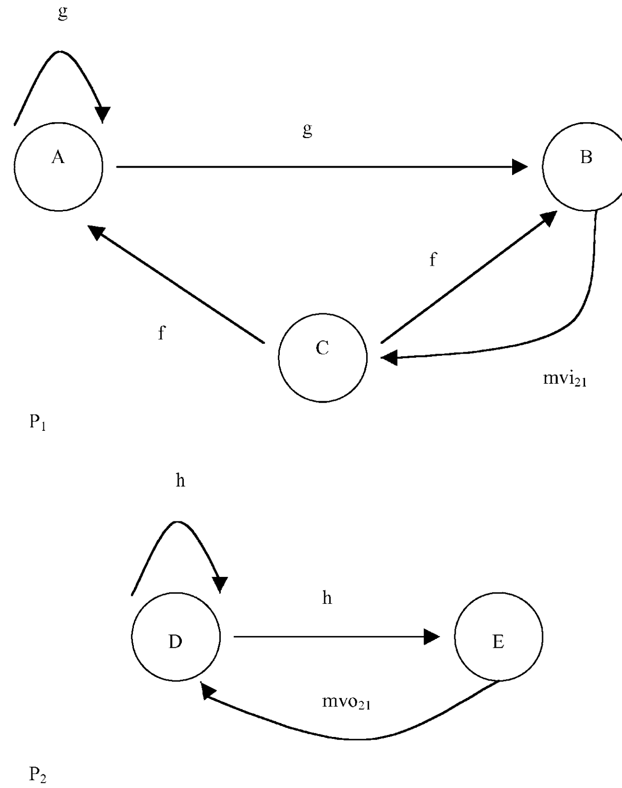

Fig. 1. Example of CSXMS

These are defined next.

$$(in_1, m_1, \lambda, x) f(y, \lambda, m_1 + in_1 - 1, \lambda), m_1 \in M_1, in_1 \in IN_1 - \{\lambda\}, m_1 + in_1 > 0$$

$$(in_1, m_1, \lambda, x) f(z, \lambda, m_1 + in_1, \lambda), m_1 \in M_1, in_1 \in IN_1 - \{\lambda\}$$

$$(\lambda, m_1, \lambda, x) g(y, \lambda, m_1 - 1, \lambda), m_1 \in M_1, m_1 > 0$$

$$(\lambda, m_1, \lambda, x) g(z, \lambda, m_1, \lambda), m_1 \in M_1$$

$$(\lambda, m_2, out_2, x) h(y, \lambda, m_2 + 1, m_2 + 1), m_2 \in M_2$$

$$(\lambda, m_2, out_2, x) h(z, \lambda, m_2, m_2), m_2 \in M_2$$

f will read the value in the input port and produce an y or a z; y's can only be produced if either the value in the input port or the internal memory of P_1 are greater than 0. g will operate only if the input port is empty; similarly, g can produce an y only if the internal memory of P_1 is greater than 0. Both f and g will update the internal memory of P_1 accordingly. h will also produce y's and z's; it will increment the internal memory of P_2 if an y has been produced and will leave it unchanged otherwise; the new value of the internal memory will also be written to the output port.

Definition 4.3. A *configuration* of a CSXMS S_n has the form

$$z = (z_1, \ldots, z_n, c),$$

where:

- $z_i = (m_i, q_i, in_i, out_i, s_i, g_i), 1 \leqslant i \leqslant n$,
- m_i is the current value of M_i, the memory of P_i,
- q_i is the current state of P_i,
- $in_i \in IN_i$ and $out_i \in OUT_i$ are the current values of the ports of P_i,

- $s_i \in \Sigma_i^*$ is the current input sequence of P_i,
- $g_i \in \Gamma_i^*$ is the current output sequence of P_i,
- c is the current communication matrix.

An initial configuration has the form $z^0 = (z_1^0, \ldots, z_n^0, c^0)$, where

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

A final configuration has the form $z^0 = (z_1^f, \ldots, z_n^f, c)$, where

$$z_i^f = (m_i, q_i^f, in_i, out_i, \epsilon, g_i) \text{ with } q_i^f \in T_i.$$

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 relation is applied.

Definition 4.4. A *change of configuration* of a CSXMS S_n, denoted by $\models$,

$$z = (z_1, \ldots, z_n, c) \models z' = (z_1', \ldots, z_n', c'),$$

with $z_i = (m_i, q_i, in_i, out_i, s_i, g_i), z_i' = (m_i', q_i', in_i', out_i', s_i', g_i')$ is possible if $z \neq z'$ and $\exists c_0, \ldots, c_n \in C$ with $c_0 = c$ and $c_n = c'$ such that for $1 \leqslant i \leqslant n$ either

- $z_i' = z_i$ and $c_i = c_{i-1}$ or
- $s_i' = \sigma_i s_i$ with $\sigma_i \in \Sigma_i \cup \{\epsilon\}$ and $g_i' = g_i \gamma_i$ with $\gamma_i \in \Gamma_i \cup \{\epsilon\}$ and $\exists \phi_i \in \Phi_i$ such that $q_i' \in F_i(q_i, \phi_i)$ and $((in_i, m_i, out_i, c_{i-1}), \sigma_i)\phi_i(\gamma_i, (in_i', m_i', out_i', c_i))$.

Note that the order chosen above is not important since the operations concerning the same location of the current matrix c are done under mutual exclusion.

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 4.5. The *relation computed* by a CSXMS, $f : (\Sigma_1^* \times \ldots \times \Sigma_n^*) \longleftrightarrow (\Gamma_1^* \times \ldots \Gamma_n^*)$, is defined by:

$(s_1 \ldots, s_n) f (g_1, \ldots, g_n) \iff \exists z^0 = (z_1^0, \ldots, z_n^0, c^0)$ and $z = (z_1, \ldots, z_n, c)$ an initial and final configurations, respectively, with $z_i^0 = (m_i^0, q_i^0, \lambda, \lambda, s_i, \epsilon)$ and $z_i' = (m_i', q_i', in_i', out_i', \epsilon, g_i')$, $1 \leqslant i \leqslant n$, such that $z^0 \models^* z$.

The behaviour of a CSXMS is not exactly that of a single SXM since there may be configuration changes that do not process any input symbol of the machine components. These configuration changes are those for which each machine component either performs a transition triggered by a communication function or no transition at all. Since the communication functions leave the input and output streams of the machine components unchanged, no change will be produced in the overall input and output of the system.

However, for testing purposes, we can introduce in the model "dummy" input and output symbols and associate them with the communication functions. In this way, any transition performed by a machine component will "consume" exactly one input symbol and "produce" exactly one output symbol so the overall system will behave like a SXM. This construction is discussed next. The way in which the addition of dummy symbols will affect the application of the SXM testing method in practice will be discussed later.

5. Testing conditions for CSXMSs

Definition 5.1. Let $S_n = ((P_i)_{1 \leqslant i \leqslant n}, C)$ be a CSXMS as above, let $a \notin (\bigcup_{i=1}^n \Sigma_i)$ and let $k_i = card(\Phi_i'')$ and $o_i : \Phi_i'' \longrightarrow \{1, \ldots k_i\}$ a one-to-one mapping, $1 \leqslant i \leqslant n$. Then $S_n^T = ((P_i^T)_{1 \leqslant i \leqslant n}, C)$ is called a *testing variant* of S_n if $P_i^T = (\Sigma_i^T, \Gamma_i^T, Q_i, M_i, \Phi_i^T, F_i^T, I_i, T_i, m_i^0, IN_i, OUT_i)$ is obtained from Pi by making the following changes, $1 \leqslant i \leqslant n$:

- The input and output alphabets are replaced by $\Sigma_i^T = \Sigma_i \cup \{a\}$ and $\Gamma_i^T = \Gamma_i \cup \{1, \ldots, k_i\}$.
- Each communication function $\phi_i'' \in \Phi''$ is replaced in the transition diagram by its testing variant, $\phi_i''^T$, defined as follows:

$$\phi_i''^T((in_i, m_i, out_i, c), a) = (o_i(\phi_i''), (in_i', m_i, out_i', c')) \text{ if } \phi_i''((in_i, m_i, out_i, c), \epsilon) = (\epsilon, (in_i', m_i, out_i', c'))$$

Thus the set of processing relations of P_i^T is

$$\Phi_i^T = \Phi_i' \cup \Phi_i''^T, \text{ where } \Phi_i''^T = \{\phi_i''^T \mid \phi_i'' \in \Phi_i\}.$$

P_i^T is called a *testing variant* of P_i.

A configuration, a configuration change and the relation computed by the testing variant of a CSXMS are defined similarly to the case of a standard CSXMS. That is, definitions 4.3 and 4.5 remain unchanged and definition 4.4 is modified as follows.

Definition 5.2. A *change of configuration of* S_n^T, denoted by $\models$,

$$z = (z_1, \ldots, z_n, c) \models z' = (z_1', \ldots, z_n', c'),$$

with $z_i = (m_i, q_i, in_i, out_i, s_i, g_i)$ and $z_i' = (m_i', q_i', in_i', out_i', s_i', g_i')$ is possible if $z \neq z'$ and $\exists c_0, \ldots, c_n \in C$ with $c_0 = c$ and $c_n = c'$ such that for $1 \leqslant i \leqslant n$ either

- $z_i' = z_i$ and $c_i = c_{i-1}$ or
- $s_i' = \sigma_i s_i$ with $\sigma_i \in \Sigma_i^T$ and $g_i' = g_i \gamma_i$ with $\gamma_i \in \Gamma_i^T$ and $\exists \phi_i \in \Phi_i^T$ with $q_i' \in F_i^T(q_i, \phi_i)$ and $((in_i, m_i, out_i, c_{i-1}), \sigma_i)\phi_i(\gamma_i, (in_i', m_i', out_i', c_i))$.

We can show now that the testing variant of a CSXMS behaves exactly like a single SXM.

Theorem 5.1. For any CSXMS S_n there exists a SXM X^T that computes the same relations as S_n^T.

Proof. We construct a stream X-machine $X^T = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m^0)$ as follows:

- $\Sigma = ((\Sigma_1^T \cup \{\epsilon\}) \times \ldots \times (\Sigma_n^T \cup \{\epsilon\})) - (\epsilon, \ldots, \epsilon)$
- $\Gamma = ((\Gamma_1^T \cup \{\epsilon\}) \times \ldots \times (\Gamma_n^T \cup \{\epsilon\})) - (\epsilon, ..., \epsilon)$
- $Q = Q_1 \times \ldots \times Q_n \times C^1$, where C^1 is the set of matrices of order $n \times n$ having as non-diagonal elements 1 or λ, where 1 is used when the corresponding communication matrix element is not empty and λ otherwise; the elements of the diagonal are all θ. Thus, if $c^1 \in C^1$, where $c^1 = (c_{ij}^1)_{1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n}$, then $c_{ij}^1 = 1$ or $c_{ij}^1 = \lambda$, $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, i \neq j$ and $c_{ij}^1 = \theta, 1 \leqslant i \leqslant n$.
  Let P_i be in state q_i, $1 \leqslant i \leqslant n$, and let c_{ij}, $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n$ be the value of the communication matrix. Then X^T will be in state $((q_1, \ldots, q_n), c^1)$, where $c_{ij}^1 = \lambda$ if $c_{ij} = \lambda$ and $c_{ij}^1 = 1$ otherwise, $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, i \neq j$.
- The set of initial states is $I = I_1 \times \ldots \times I_n \times \{c^0\}$, where $c^0 = (c_{ij}^0)_{1 \leqslant i \leqslant n}$ is the matrix with all non-diagonal elements λ, i.e. $c_{ij}^0 = \lambda$, $1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, i \neq j$ and $c_{ii}^0 = \theta, 1 \leqslant i \leqslant n$.
- The set of final states is $T = T_1 \times \ldots \times T_n \times C^1$.
- $M = (IN_1 \times M_1 \times OUT_1) \times \ldots \times (IN_n \times M_n \times OUT_n) \times C$. The global memory contains the memory values, input and output ports of each component M_i as well as the communication matrix. The initial memory value is $m^0 = ((m_1^0, \lambda, \lambda) \ldots, (m_n^0, \lambda, \lambda), c^0)$, that contains all the initial memory and port values as well as the initial communication matrix.
- $\Phi = \{(\phi_1, \ldots, \phi_n) \in ((\Phi_1 \cup \{e\}) \times \ldots \times (\Phi_n \cup \{e\})) \mid (\phi_1, \ldots, \phi_n) \neq (e, \ldots, e)$ and $(\neg \exists i, j, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, i \neq j$, such that $\phi_i = mvo_{ij}$ and $\phi_j = mvi_{ij})\}$. A relation $(\phi_1, \ldots, \phi_n) : M \times \Sigma \longrightarrow \Gamma \times M$ stands for all components $\phi_i \in \Phi_i^T \cup \{e\}$ acting in parallel. $\phi_i = e$ denotes that P_i does not change its configuration. More formally

$$(\phi_1, \ldots, \phi_n)(((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n), c), (\sigma_1, \ldots, \sigma_n)) =$$

$$(((in_1', m_1', out_1'), \ldots, (in_n', m_n', out_n'), c'), (\gamma_1, \ldots, \gamma_n))$$

  if $\exists c_0, \ldots, c_n \in C$ with $c_0 = c$ and $c_n = c'$ such that for $1 \leqslant i \leqslant n$ either

  - $in_i' = in_i, m_i' = m_i, out_i' = out_i, \sigma_i = \gamma_i = \epsilon$ and $c_i = c_{i-1}$ or
  - $s_i' = \sigma_i s_i$ with $\sigma_i \in \Sigma_i^T$ and $g_i' = g_i \gamma_i$ with $\gamma_i \in \Gamma_i^T$ and $\exists \phi_i \in \Phi_i^T$ with $q_i' \in F_i^T(q_i, \phi_i)$ and $((in_i, m_i, out_i, c_{i-1}), \sigma_i)\phi_i(\gamma_i, (in_i', m_i', out_i', c_i))$

- The transition function $F : Q \times \Phi \longrightarrow 2^Q$ is defined as follows:

$$(q'_1, \ldots, q'_n, c^1) \in (F((q_1, \ldots, q_n), (\phi_1, \ldots, \phi_n)), c'^1) \text{ if for } 1 \leqslant i \leqslant n \text{ either}$$

  - $\phi_i = e$ and $q'_i = q_i$ or
  - $\phi_i \in \Phi'_i$ and $q'_i \in F_i(q_i, \phi_i)$ or
  - $q'_i \in F_i(q_i, \phi_i)$ and $\phi_i = mvo^T_{ij}, c^1_{ij} = \lambda$ and $c'^1_{ij} = 1$ for some $j, 1 \leqslant j \leqslant n, j \neq i$ or
  - $q'_i \in F_i(q_i, \phi_i)$ and $\phi_i = mvi^T_{ji}, c^1_{ji} = 1$ and $c'^1_{ji} = \lambda$ for some $j, 1 \leqslant j \leqslant n, j \neq i$.

  The rest of the elements of the matrix c^1 (those that are not mentioned explicitly above) will remain unchanged.
  Thus, the transition function of X^T will reflect the transitions of the machine components. Also, a non-diagonal element c^1_{ij} of the matrix c^1 will change from λ to 1 if mvo_{ij} writes to the (i, j) location of the communication matrix and from 1 to λ if mvi_{ij} reads from the (i, j) location of the communication matrix.

X^T is called the *equivalent SXM* of S^T_n.

A configuration of X^T is, according to definition 2.2, a tuple

$$\zeta = ((m_1, in_1, out_1), \ldots, (m_n, in_n, out_n), c, (q_1, \ldots, q_n), c^1, (s_1, \ldots, s_n)(g_1, \ldots, g_n)),$$

where c^1 is obtained from c by replacing all non-empty non-diagonal elements with 1, i.e.

- $c^1_{ij} = \theta$ if $c_{ij} = \theta$
- $c^1_{ij} = \lambda$ if $c_{ij} = \lambda$
- $c^1_{ij} = 1$ if $c_{ij} \in C_{ij} - \{\lambda, \theta\}$

From the above construction it follows directly that X^T works exactly like S^T_n since:

$$\zeta \vdash \zeta' \Longleftrightarrow z \models z',$$

where

$$z = ((m_1, q_1, in_1, out_1, s_1, g_1), ..., (m_n, q_n, in_n, out_n, s_n, g_n), c)$$

represents the configuration of S^T_n. □

The equivalent SXM of the CSXMS in example 4.1 may be found in figure 2. The state set is

$$Q = \{A, B, C\} \times \{C, D\} \times \{\lambda, 1\},$$

where λ and 1 indicate whether the location $(2, 1)$ of the communication matrix is empty or not. Note that the location $(1, 2)$ is always empty so it will not affect the construction of the state set of the equivalent SXM.

Lemma 5.1. If the associated automaton of P_i is deterministic, $1 \leqslant i \leqslant n$, then the associated automaton of X^T is also deterministic.

Proof. Follows from the construction of F. □

Lemma 5.2. If Φ''_i is a set of (partial) functions (rather than relations) then Φ is also a set of (partial) functions.

Proof. If Φ'_i is a set of (partial) functions then Φ^T_i is also a set of (partial) functions, thus Φ is a set of (partial) functions. □

The remaining part of this section investigates how the "design for testing" properties of the individual CSXM components are reflected in similar properties of the resulting SXM, X^T. That is, if the CSXM components meet a "design for test condition", will the resulting machine X^T also meet the same property or at least its relaxed variant (definitions 3.4 and 3.5). We prove that the implication is true in the case of output-distinguishability property and identify a class of CSXM for which the implication is also true in the case of the completeness property.

Legend.

x = mvi $_{21}$ y = mvo $_{21}$

Fig. 2. The equivalent SXM of the CSXMS example

Theorem 5.2. Let $V_i \subseteq M_i, V'_i = IN_i \times V_i \times OUT_i$ and $U_i = \{U_i^{\phi_i} \mid \phi_i \in \Phi'_i\}$ with $U_i^{\phi_i} \subseteq \Sigma_i$ such that Φ'_i is closed w.r.t. $(V'_i, U_i), 1 \leqslant i \leqslant n$. Then Φ is closed w.r.t. (V, U), where

- $V = V_1 \times \ldots \times V_n \times C$
- $U = \{U^\phi \mid \phi \in \Phi\}$, where $U^\phi = (U_1^{\phi_1}, \ldots, U_n^{\phi_n})$ with $\phi = (\phi_1, \ldots, \phi_n)$, where
  - $U_i^{\phi_i} = \{a\}$ if $\phi_i \in \Phi_i''^T$,
  - $U_i^e = \{e\}$

Proof. Let $m = ((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n), c) \in V$, $m' = ((in'_1, m'_1, out'_1), \ldots, (in'_n, m'_n, out'_n), c') \in M$, $\phi = (\phi_1, \ldots, \phi_n) \in \Phi$, $\sigma = (\sigma_1, \ldots, \sigma_n) \in U^\phi$, $\gamma = (\gamma_1, \ldots, \gamma_n) \in \Gamma$ such that $(m, \sigma)\phi(\gamma, m')$ If $\phi_i \in \Phi'_i$ then $m'_i \in V_i$ since Φ'_i is closed w.r.t. (V'_i, U_i). If $\phi_i \in \Phi_i''^T$ or $\phi_i = e$ then $m'_i = m_i$. Thus $m'_i \in V_i$, $1 \leqslant i \leqslant n$, hence $m' \in V$. □

Theorem 5.3. If Φ'_i is output-distinguishable w.r.t. (V'_i, U_i), $1 \leqslant i \leqslant n$, then Φ is output-distinguishable w.r.t. (V, U).

Proof. Let us consider two arbitrary functions $\phi = (\phi_1, \ldots, \phi_n), \psi = (\psi_1, \ldots, \psi_n) \in \Phi$ and suppose that there exist $m = ((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n), c) \in V$, $(\sigma_1, \ldots, \sigma_n) \in U^\phi \cap U^\psi$ such that $(m, \sigma)\phi(\gamma, m')$ and $(m, \sigma)\phi(\gamma, m'')$ for some $m' = ((in'_1, m'_1, out'_1), \ldots, (in'_n, m'_n, out'_n), c') \in V$, $m'' = ((in''_1, m''_1, out''_1), \ldots, (in''_n, m''_n, out''_n), c'') \in V$ and $\gamma = (\gamma_1, \ldots, \gamma_n) \in \Gamma$. Then there exist $c'_0, \ldots, c'_n, c''_0, \ldots, c''_n \in C$ with $c'_0 = c''_0 = c, c'_n = c', c''_n = c''$ such that $((in_i, m_i, out_i, c'_{i-1}), \sigma_i)\phi_i(\gamma_i, ((in'_i, m'_i, out'_i, c'_i))$ and $((in_i, m_i, out_i, c''_{i-1}), \sigma_i)\phi_i(\gamma_i, ((m''_i, in''_i, out''_i, c''_i))$, $1 \leqslant i \leqslant n$. ¿From the output-distinguishability property of Φ'_i and the construction of $\Phi_i''^T$ if follows that $\phi_i = \psi_i$, $m'_i = m''_i$, $in'_i = in''_i$, $out'_i = out''_i$, $c'_i = c''_i$, and so $\phi = \psi$ and $m' = m''$. □

Definition 5.3. A CSXM P_i is called simple if the following requirements are met:

- All initial states are ordinary states, i.e.
$$I_i \subseteq Q'_i.$$
- An ordinary relation always empties the input port; furthermore, an ordinary relation always produces a non-empty value for the output port, provided that such values are allowed, i.e. $\forall \phi'_i \in \Phi'_i$
$$((in_i, m_i, out_i), \sigma_i)\phi'_i(g_i, (in'_i, m'_i, out'_i)) \Longrightarrow in'_i = \lambda \text{ and } (out'_i \neq \lambda \text{ or } OUT_i = \{\lambda\}).$$
- A communication function always takes the machine in an ordinary state, i.e. $\forall q''_i \in Q'', \phi''_i \in \Phi''_i$,
$$F_i(q''_i, \phi''_i) = p_i \Longrightarrow p_i \in Q'_i.$$

A Communicating system $S_n = ((P_i)_{1 \leqslant i \leqslant n}, C)$ is called simple if P_i is simple, $1 \leqslant i \leqslant n$.

For instance, the CSXMS of example 4.1 is simple.

Lemma 5.3. Let $S_n = ((P_i)_{1 \leqslant i \leqslant n}, C)$ be a simple CSXMS and let $\phi = (\phi_1, \ldots, \phi_n) \in \Phi$ and $m = ((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n)), c) \in M$ such that $m \in Mattain_\phi(P)$. If $\phi_i \in \Phi''_i$ then $in_i = \lambda$ and $(out_i \neq \lambda$ or $OUT_i = \{\lambda\})$, $1 \leqslant i \leqslant n$.

Proof. Since $m \in Mattain_\phi(P)$, $\exists q = (q_1 \ldots, q_n, c) \in Q$ such that $m \in Mattain_q(P)$. Let $1 \leqslant i \leqslant n$ such that $\phi_i \in \Phi''_i$. Then from the construction of X^T it follows that $q_i \in Q''_i$. Now, since S_n is simple q_i is not an initial state of M_i and all arcs of M_i that enter q_i are labelled by ordinary relations. Since $m \in Mattain_q(P)$ it follows that $in_i = \lambda$ and $(out_i \neq \lambda$ or $OUT_i = \{\lambda\})$. □

Theorem 5.4. If S_n is simple and Φ'_i is complete w.r.t. (V'_i, U_i), $1 \leqslant i \leqslant n$, then X^T is input-complete w.r.t. (V, U).

Proof. Let $\phi = (\phi_1, \ldots, \phi_n) \in \Phi$ and $m = ((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n), c) \in Mattain_\phi(P) \cap V$. Let $1 \leqslant i \leqslant n$. We have the following cases:

- $\phi_i \in \Phi'_i$. Since $(in_i, m_i, out_i) \in V'_i$ then $\exists \sigma_i \in U_i^{\phi_i}$ such that $\phi_i((in_i, m_i, out_i, c), \sigma_i) \neq \emptyset$.
- $\phi_i = \phi_i''^T$ for some $\phi_i \in \Phi_i''^T$. Then from the construction of X^T it follows that either:
  - $\phi''_i = mvo_{ij}$ and $c_{ij} = \lambda$ or
  - $\phi''_i = mvi_{ji}$ and $c_{ji} \neq \lambda$

  for some j, $1 \leqslant j \leqslant n$, $j \neq i$. Also, since $m \in Mattain_\phi(P)$ and $\phi''_i \in \Phi''_i$, from the above lemma it follows that $in_i = \lambda$ and $(out_i \neq \lambda$ or $OUT_i \neq \{\lambda\})$. Thus, either
  - $\phi''_i = mvo_{ij}, c_{ij} = \lambda$ and $out_i \neq \lambda$ or
  - $\phi''_i = mvi_{ji}, c_{ji} \neq \lambda$ and $in_i = \lambda$.

  Thus, in either case, $\phi''_i((in_i, m_i, out_i, c), \epsilon) \neq \emptyset$, hence $\phi_i((in_i, m_i, out_i, c), a) \neq \emptyset$.
- $\phi_i = e$. Then $\phi_i((in_i, m_i, out_i, c_i), \epsilon) \neq \emptyset$. Hence $\exists \sigma_i \in U_i^\phi$ such that $\phi((in_i, m_i, out_i, c), \sigma_i) \neq \emptyset$, $1 \leqslant i \leqslant n$. Thus $\phi(m, \sigma) \neq \emptyset$, where $\sigma = (\sigma_1, \ldots, \sigma_n) \in U^\phi$.

□

6. Application of the SXM testing method to a CSXMS

The previous section identifies a number of conditions that each machine component has to meet in order to ensure that the design for test conditions (or at least their relaxed variants) of the machine components are preserved at system level. There are three such conditions:

- The initial states of all machine components are ordinary states. That is, each machine has to perform some action before the communication can begin.
- The application of any ordinary relation will always empty the input port and produce a non-empty value for the output port (if this exists). These requirements are quite natural and can be easily enforced by the designer. For example, it is natural to assume that an ordinary relation will process the value in the input

port if this has not been processed yet; when this is not true, the relation can be modified so that the input port value is collected and stored in the memory for later use.

- The application of a communication function will always take a machine component to an ordinary state, thus no machine component can perform more than one consecutive communicating action. In general, any communicating machine whose associated automaton does not contain loops of communication functions can be re-designed to meet this condition (in this case there are only finite sequences of communication functions and these can be composed to produce the communication functions of the re-designed machine). This condition is crucial for testing since it disallows potentially infinite sequences of communication actions that cannot be controlled by the tester.

Furthermore, our construction introduced dummy input and output symbols so the natural question to ask is how these extra symbols are represented in practice. The dummy outputs are just means of distinguishing between communication functions and in practice this is not usually hard to achieve. Things are slightly more complicated in the case of the dummy input: it will be awkward and unnatural to associate inputs to the processing functions and to expect these functions to be triggered by such external events. However, this is not necessary, as explained bellow.

First, let us establish some notation. We will use S to refer to the real system under testing and S^T to the testing variant of our specification; S^T uses a dummy input symbol to trigger communication functions, whereas S performs silent moves (i.e. moves that consume no input) instead. Let us denote by Σ and Σ^T the input alphabets of S and S^T, respectively. Thus

$$\Sigma^T = ((\Sigma_1^T \cup \{\epsilon\}) \times \ldots \times (\Sigma_n^T \cup \{\epsilon\})) - (\epsilon, \ldots, \epsilon)$$

and

$$\Sigma = ((\Sigma_1 \cup \{\epsilon\}) \times \ldots \times (\Sigma_n \cup \{\epsilon\})) - (\epsilon, \ldots, \epsilon),$$

where Σ_i is the input alphabet of the i-th CSXM component and $\Sigma_i^T = \Sigma_i \cup \{a\}$, $1 \leqslant i \leqslant n$. For $1 \leqslant i \leqslant n$, let $filter_i : \Sigma_i^{T*} \longrightarrow \Sigma_i^*$ be the free-semigroup morphism induced by

$$filter_i(\sigma_i) = \sigma_i, \sigma_i \in \Sigma_i,$$

$$filter_i(a) = \epsilon,$$

and let $filter = (filter_1, \ldots, filter_n)$, $filter : \Sigma^{T*} \longrightarrow \Sigma^*$.

Note that S may be non-deterministic even when all the CSXM components of S^T are deterministic (in the sense explained in section 2); this is because a machine component P_i that is in a communication state, say q_i'', may remain in the current state or may use a communication function to move to another state while the other machine components can change their configurations. So, the communication functions not only allows the system to perform silent moves but also introduces non-determinism.

In order to test non-deterministic systems, one usually makes a so-called *complete-testing assumption* [LBP94]: it is possible, by applying a given input sequence s to the system a finite number of times, to exercise all the paths of the system that can be traversed by s. Without such an assumption, no test suites can guarantee full fault coverage for non-deterministic systems. Obviously, such an assumption can only be made if the number of paths that can be traversed by s is *finite*.

Let us see how this complete-testing assumption works in our case. Let $s_0^T \in \Sigma^T$ be a test sequence generated from our specification S^T. The application of this sequence to the specification corresponds to the application of the sequence $s = filter(s_0^T)$ to S. On the other hand, S may be non-deterministic, so s may trigger S to traverse a number of paths, these correspond to paths in S^T exercised by sequences in $filter^{-1}(s)$. Now, we have established that no CSXM component can execute more than one consecutive communication function, so the corresponding paths in S^T will be those exercised by inputs in

$$Y_s = \{s^T \in filter^{-1}(s) \mid length(s^T) \leqslant (n+1) \cdot length(s)\}$$

Since Y_s is finite we can assume that it is possible, by applying s to S a sufficient number of times, to exercise the path triggered by s_0^T in S^T. Obviously, the quality of testing increases with the number of repetitions of test sequence application; in actual testing, this number is limited by practical and economical considerations.

7. Conclusions

SXMs are generalisations of FSMs similar to EFSMs. The current approach to EFSM testing [LeY96], [ChK93], [WaL93] is a straightforward application of the testing methods for FSMs: given an EFSM, if each variable has a finite number of values then there is a finite number of configurations (tuples of states and variable values) and there is an equivalent FSM with configuration as states, so testing EFSMs reduces in principle to testing of FSMs. However, in many practical applications, the equivalent FSM may have such a large number of states that it is impossible to even construct it, let alone to derive a test set of a reasonable size. This is the well known state explosion problem. There are a number of approaches to cope with this [ChK93], [ChA90], [Kwa92], [LeL91], [WaL93] the general idea is to "collapse" equivalent states into one to compute a reduced machine and to select from the reduced machine only the states that are reachable from the initial state [LeY96].

The approach employed by the SXM testing method is rather different. Here, a number of transition are grouped into processing functions or relations and these are assumed to be implemented correctly. Thus, testing the SXM reduces to testing its associated automaton. The correctness of the processing functions or relations is checked by a separate process: depending on the nature of the function or relation, this can be tested using the same method or alternative functional methods [HoI98], [IpH98b], [HIG95]. Furthermore, the method can only be applied if the processing functions or relations meet some "design for test conditions", completeness and output-distinguishability. On the other hand, the SXM testing method does not rely on the memory set being finite (since there is no need to construct the equivalent finite state machine) and furthermore, it avoids the state explosion problem.

Communication between different processes can often be modelled as a collection of communicating finite state machines (CFSM) [LeY96] or extended communicating finite state machines (ECFSM) [HuH94]. For testing purposes, we can first take all combinations of states, variables and communication channels and construct a composite machine, which is a FSM if all these variables and channels have a finite number of values, and then apply known techniques to the composite machine. Again, we may run into the state explosion problem, especially in the case of ECFSMs, and this can be relieved using reduction techniques [LCL187], [LCL287], [HuH94].

Up to a point, our approach to testing CSXMSs is similar: we construct an equivalent SXM that is equivalent to the whole system. In order to do this, we use a dummy input to trigger the communication functions of the CSXM components. A state of the equivalent SXM will consists of a combination of the states of the machine components and a matrix of order $n \times n$ having Boolean values as non-diagonal elements. The state explosion problem may still appear, but this will be much more limited than in the case of ECFSMs since the memory values of the components are not used in the construction of the equivalent SXM. Existing minimisation techniques for FSMs will still be used to obtain the minimal automaton of the equivalent SXM.

Our paper goes on to investigate to what extent the equivalent SXM can preserve the "design for test conditions" of the CSXM components and identifies a class of CSXMs for which the equivalent SXM of the system meets these "design for test conditions" (more precisely, their relaxed variants).

The CSXM model defined in [BCG99] and investigated in this paper can offer only a low level of synchronisation. More recently, [GeV00] defines a new CSXM model that uses channels as a higher level of synchronisation. A future paper will investigate the application of the SXM to this new model.

References

[BCG99] T. Balanescu, T. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, C. Vertan, Communicating Stream X-Machines Systems are no more than X-Machines, *J.UCS*, 5 (9), pp. 492-507, 1999.

[ChK93] K.-T. Cheng, A.S. Krishnakumar, Automatic functional test generation using the extended finite state machine mode, *Proc. DAC*, pp 86–91, 1993.

[Cho78] T. S. Chow, Testing software design modelled by finite state machines, *IEEE Transactions on Software Engineering,* 4(3), pp. 178–187, 1978.

[ChA90] W. Chun, P.D. Amer, Test Case Generation For Protocols Specified In Estelle, FORTE'90, Madrid, Spain, J. Quemada, J. Manas, E. Vazquaz (eds) pp. 197–210, 1990.

[Eil74] S. Eilenberg, *Automata, languages and machines,* Vol. A, Academic Press, 1974.

[FHI95] M. Fairtlough, M. Holcombe, F. Ipate, C. Jordan, G. Laycock, Z. Duan, Using an X-machine to Model a Video Cassette Recorder, *Current issues in electronic modelling,* 3, pp/ 141-161, 1995.

[FBK91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, Test Selection Based on Finite State Models, *IEEE Transactions on Software Engineering,* 17(6), pp. 591-603, 1991.

[GeV00] H. Georgescu, C. Vertan, A New Approach to Communicating Stream X-Machines Systems, accepted by *J.UCS*, 2000.

[Hol88] M. Holcombe, X- machines as a basis for dynamic system specification, *Software Engineering Journal* 3, pp69-76, 1998.

[HIG95] M. Holcombe, F. Ipate, A. Grondoudis, Complete Functional Testing of Safety-Critical Systems, *Safety and Reliability in Emerging Control Technologies: A Postprint volume from the IFAC Workshop on Safety and Reliability in Emerging Control Technologies*, Daytona Beach, Florida, USA, pp. 199–204, 1-3 November 1995.

[HoI98] M, Holcombe and F. Ipate, *Correct Systems: Building a Business Process Solution,* Springer Verlag, Berlin, 1998.

[HuH94] C.-M. Huang, J.-M. Hsu, An incremental Protocol Verification Method, *The computer Journal*, 37(8), pp. 698–710, 1994.

[IpH96] F. Ipate and M. Holcombe, Another look at computability, *Informatica,* Vol. 20, pp. 359–372, 1996.

[IpH97] F. Ipate and M. Holcombe, An Integration Testing Method That is Proved to Find all Faults, *Intern. J. Computer Math,* Vol. 69, pp. 159–178, 1997.

[Ipa95] F. Ipate, *Theory of X-machines and Applications in Specification and Testing,* PhD thesis, University of Sheffield, UK, 1995.

[IpH98a] F. Ipate, M. Holcombe, *A method for refining and testing generalised machine specifications, International Journal of Computer Mathematics,* 68, pp. 197–219, 1998.

[IpH98b] F. Ipate, M. Holcombe, Specification and Testing using Generalised Machines: a Presentation and a Case Study, *Software Testing, Verification and Reliability,* 8, pp. 6–81, 1998.

[IpH01] F. Ipate, M. Holcombe, Generating Test Sequences from Non-deterministic Generalised Stream X-machines, *Formal Aspects of Computing* 12:443–458, 2000.

[Kwa92] E. Kwast, Automatic test generation for protocol data aspects, *Proc. IFIG WG6.1 12th Intl. Symp. on Protocol Specification, Testing and Verification*, North Holland, R.J. Linn, Jr. and M.U.Uyar Ed., pp 211-226, 1992.

[LeL91] D.Y. Lee, J.D. Lee, Test generation for the specification written in Estelle, *Proc. IPIF WG6.1 11th Intl. Symp on Protocol Specification, Testing and Verification*, North-Holland, B. Jonsson, J. Parrow, B. Pehrson Ed., pp 317-332, 1991.

[LeY96] D. Lee, M. Yannakakis, Principles and Methods of Testing Finite State Machines - A Survey, *Proceedings of the IEEE*, 84(8), pp. 1090-1123, 1996.

[LCL187] F. Lin, P. Chu, M. Liu, Protocol Verification using Reachability Analysis, *Computer Communication Review*, Vol. 17, No. 5, 1987.

[LCL287] F. Lin, P. Chu, M. Liu, Protocol Verification using Reachability Analysis: the state explosion problem and relief strategies, *Computer Communication Review*, 17(5) 126–134, 1987.

[LBP94] G. Luo, G. v. Bochmann, A. Petrenko, Test Selection Based on Communicating Non-deterministic Finite-State Machines Using a Generalised Wp-Method, *IEEE Transactions on Software Engineering,* 20(2), pp. 149–161, 1994.

[WaL93] C.-J. Wang, M.T. Liu, Generating test cases for EFSM with given fault models, *Proc. IEEE INFOCOM'93*, pp 774–781, 1993.

Received November 1999
Accepted in revised form June 2001 by C B Jones