Systems of Communicating X-machines for Specifying Distributed Systems

Research Report CS-02-07

J. Aguado and A. J. Cowling

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

Abstract

Various approaches have been proposed to the problem of assembling X-machines (also known as Eilenberg X-machines) into a communication system. In this report, these approaches are presented and unified within the stand-alone X-machine notation. The models are analysed highlighting those aspects that seem to be more relevant for specifying distributed (testable) systems. From the testing perspective, it has been proved that the Holcombe-Ipate testing approach (SMXT), developed originally for stream X-machines, can be applied to some of these communicating systems. For one of the approaches, the CXM-system, the formalism needs to be modified if the testing method will be used and these modifications are discussed. Another of these models, the CSXMS, is surveyed and all its variations are studied in order to provide the necessary conditions for testing it. A different model, the CSXMS-c, that allows a synchronous mechanism for message-passing, is also analysed. The results of this show the correct implementation of the construct and the passing of messages.

A methodology for building communicating X-machines from stand-alone X-machines is also included in this report. This methodology, the MSS, is approached here by means of a modified version of the multiple-stream X machines (M-SXM). These systems, the CM-SXMS, are defined in terms of a graph, where the vertices model the components, and the edges correspond to streams that are shared between them. It seems that the CSXMS-c, CSXMS and CM-SXMS can respectively model the distributed computing models of synchronous, semi-synchronous and asynchronous message-passing. Therefore, if the SXMT can be extended and applied to all of the communicating X-machine systems, then it could be possible to test distributed algorithms with different message-passing structures, but this will require future work

Keywords: X-machines, communicating X-machines, communicating stream X-machines systems (CSXMS), CSXMS testing-variant, simple CSXMS, CSXMS with channels, modular specification of systems using communicating X-machines, communicating multiple-stream X-machines systems, formal specification, distributed systems, testing.

Contents

1. Introduction

The approach that is studied here is based on the *X-machine* model, which was originally introduced by Eilenberg [1] in 1974 as an alternative to *finite-state machines* (FSM), *pushdown automata* (PDA) *Turing machines* (TM) and other kinds of machine model. It was not until 1988, however, that this form of abstract model was used as a possible specification language by Holcombe [2]. Since then, much work has been done and applied to a large number of specification and modelling problems, e.g. in [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. Note that these mechanisms have been referred to in the literature as X-machines, as they were originally named by Eilenberg, it has been suggested (reported in [24]) their name to be changed to *Eilenberg machines*. Therefore, the name (Eilenberg) X-machine will be used throughout this report.

There are several generalisations and extensions of the original model [4, 6, 10, 25, 26] and they are reviewed and compared in [27]. However, our main interest here is in the new proposals for providing communication capabilities for the EXM [28, 29, 30, 31, 32, 33, 34, 35]. The reason for our interest in these *communicating (Eilenberg) X-machines* models is that it seems possible to specify *distributed systems and algorithms* in this way.

In this report we use the term distributed system to mean a set of autonomous computational units (processors, processes or machines) that have processing and storage capabilities and that are interconnected by an arbitrary structure of communication. Such units can be "programmed" to exchange messages through the communication media (e.g. channels) or to execute local computations. We shall use the term distributed-algorithm to mean the aggregation of a set of algorithms running in the diverse computational units of a distributed system to find a common solution for a particular problem. It is possible to abstract two aspects inherent in these concepts. One is a *structural* component, that has to do with the elements and the (communication) interrelations among them, and the other is a *dynamic* component, that corresponds to the states and changes of state that occur in the system (behaviour). With respect to the latter, it can be established that the global state of a distributed algorithm is the set of local states of the processes and the state of the communication media at a given time. The local states can be represented naturally by the EXM formalism since the data space is independent of the control structure and thus we can model both. The state of the media can be defined as the set of messages in transit. Clearly, different classes of EXM can be employed and diverse underlying communication structures can be proposed, resulting in the various models found in the literature. As may be expected, different authors not only employ diverse notations but also associate different meanings and intentions to the concept of communication for each model.

Even more important for choosing communicating EXM systems as a central theme of study here, is that distributed systems and algorithms pose particular problems for testing, as the high-degree of non-determinism means that errors are not easily detected. Consequently, traditional testing techniques are of limited use for them [36]. However, some recent investigations have shown that the Holcombe-Ipate testing approach originally developed for SXM (SXMT) [6, 10, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51] can be extended to some systems of communicating EXM models [31, 33].

The SXMT in its simplest form involves identifying test cases from the different paths that the machine can take through its set of control-states. The method used for this, which is based on the Chow W-method [52] and Fujiwara Wp-method [53], depends only on the control-states, and is independent of the different possible memory-states. Thus, it produces a much smaller test set than would be possible with any method that had to consider both sets of states. Furthermore, the process of constructing this test set also identifies the *design-for-test* conditions that need to be fulfilled [10, 43]. If these constraints are satisfied, it has been demonstrated that the test set which is generated corroborates the functional equivalence between SXM behaviour (*i.e.* find all faults) [43]. The SXMT method was originally developed for deterministic machines. However, it has recently been extended and modified to deal with non-determinism [54, 55, 56, 57, 58, 59].

Hence, this report is intended as a reference guide to the different versions, which have been proposed for the communicating EXM model, emphasising in particular the concepts, and techniques that are relevant to specifying and testing distributed systems and algorithms. In doing this, it is an objective to establish a regularity of mathematical notation, both between the variants of this model, and with the notation that has been used in describing stand-alone EXM [27].

The next section overviews the general approach to this and indicates how the rest of this report is organised.

2. Overview

A basic model for assembling EXM into a communication system was introduced originally by Barnard *et al.* [28] in 1996. The *communicating (Eilenberg) X-machines system* (CXM-system) that they described is based on EXM with input and output ports, where each output port of one machine might be connected to an input port of another machine by a channel. Section 3 presents this CXM-system model and, in particular, section 3.1 gives its basic definitions. Unfortunately, from the functional testing perspective, the CXM-system was not developed to a point where the *input-output relationship* could be derived from it (*i.e.* the output corresponding to an input sequence). Related to this, in section 3.2 we analyse the problems associated with identifying the class of EXM on which this system is supported, basing our analysis on the EXM operation. The main conclusion of our investigation on this is that there is not a restricted and clear form of interaction with the environment (*i.e.* outside the boundaries of the system). Thus, this formalism needs to be augmented if the input-output relationship is required, which is the case if the SXMT method is to be employed. Finally, in section 3.3 a farmer process example is completely specified using this CXM-system.

The basic model for communicating EXM was modified later in 1999 by Balanescu *et al.* [31] who instead modelled the communication mechanism in terms of a *communication matrix* and represented the system components as *stream X-machines* (SXM) [4, 25] to give a model known as the *communicating (Eilenberg) stream X-machines systems* (CSXMS). An important advantage of this idea is that it defines how the input-output relationship can be obtained. Section 4 explains the general ideas of the model and in section 4.1 the CSXMS is precisely formalised with three concepts deserving particular attention:

1. The set of (partial) functions (*i.e.* type) of the components of a CSXMS is formed by two disjoint subsets, namely the set of *processing functions* and the set of *communicating functions*. The notion is highly intuitive, the processing functions perform internal computations of a given component whilst the communicating functions send or receive messages.
2. The set of finite states of the components of a CSXMS is partitioned into two disjoint subsets of *processing states* and *communicating states*. All the transitions emerging from a processing state or a communicating state correspond to processing or communicating functions respectively.
3. Each component has just one output port and one input port for interchanging messages. Communicating functions indicate where the information from the output port will be sent or from which component the input port will receive the information.

As a comparative example, section 4.2 contains the specification of the same farmer process as section 3.3. Section 4.3 then extends the analysis to formalise the notions of changes in configuration and the input-output relation for this model. We introduce the concept of single change of configuration, as a way of reasoning about the system in terms of individual steps, in order to provide an alternative explanation of the property of concurrence in the context of CSXMS behaviour. Of particular interest is the consideration in section 4.4 of the transformation from a CSXMS into a functionally equivalent EXM following the ideas of [31]. This in principle admits the application and extension of the existing testing strategy for SXM. In general, the process of testing from a CSXMS will consist of constructing an equivalent EXM and then applying the testing method to it. Nevertheless, this algorithm could produce a machine that supports *empty-operations* [33, 60]. Technically speaking, an empty-operation is a transition (i.e. function or relation) of the machine that leaves the input and the output streams unchanged. In the strictest sense those machines are *(Eilenberg) straight-move stream X-machines* (Sm-SXM). Independently from the fact that the SXM is a more restricted class than the Sm-SXM, an important observation is that the SXMT requires detectable behaviour, which is in principle impossible to obtain when no input and no output can be observed.

The above difficulty was solved by Ipate *et al.* [33] via a conversion process that modifies a CSXMS in such a way that it is guaranteed that any transition performed by the overall system consumes and produces no more than one input and one output symbol. Therefore, this modification labelled the CSXMS *testing-variant* denoted by $CSXMS_n^T$ behaves like an SXM. The testing variant notions are reviewed in section 5: specifically, section 5.1 is devoted completely to the formalisation of the algorithm of modification and following [33] it is described the proof of the existence of an equivalent SXM for any $CSXMS_n^T$. In other words, for any CSXMS, there is a SXM that computes exactly the same relation as $CSXMS_n^T$.

A far more challenging problem than the input-output detectability (i.e. no empty-operations are allowed), is that the SXMT ensures a reliable and complete test set if certain constraints are met by the system [6, 10, 38]. These *design-for-test* conditions are *completeness* and *output-distinguishability*. Consequently, in order to employ successfully the SMXT for CSXMS it is necessary to ensure that the design-for-test conditions exist at system

level. In the case of output-distinguishability it has been shown [33] that when it is present in all the components it is also inherited by the $CSXMS_n^T$, which is shown in section 5.2. The case of completeness is treated separately in section 5.3 where a class called the *simple* CSXMS (CSXMS-s) identified in [33] is defined. The important property of the CSXMS-s is that the resulting SXM meets *input-completeness* (i.e. the relax-variant of completeness) when completeness is present in all the components of the system. This section therefore explains that a CSXMS-s is a CSXMS that satisfies the following three conditions:

1. The initial states of all the components of the system are processing states. Put differently, no component in the system will start its execution by performing a communication.
2. Every processing function always empties the input port and produces a non-empty value for the output port.
3. Every communicating function emerges from a communicating state and takes the component to a processing state.

A different version for communicating EXM was proposed in [34]. This model allows the use of channels as the basic mechanism for exchanging messages and offers a higher level of synchronisation with respect to the CSXMS. The operations of communication are done by *rendezvous*; that is to say, when a message is passed between machines the first machine ready to communicate is blocked until the complementary machine is also ready to complete the message transaction. This CSXMS with channels model (CSXMS-c) is reviewed in section 6. The presentation of the model is addressed in section 6.1, which explains that any CSXMS-c includes an additional communicating machine called the *communication server*. The precise purpose of the server is to control the synchronisation for the exchange of messages. Thus, a protocol or *macro-function* is performed to control a send/receive operation. This protocol occurs among the machines that are trying to establish the communication and the server. Accordingly, the server recognises the requests from the components for sending or receiving messages, and depending on the conditions and the state of the system, the server either authorises or rejects the operation. In order to achieve this, the type of communication permitted between the server and any of the components of the system has been modified. In effect, the matrix structure of the communication subsystem of a CSXMS allows the modelling of full-duplex channels (i.e. the communication is bi-directional and can occur simultaneously) between any two components. Nevertheless, the strategies of communication with the server in the CSXMS-c are based on defining half-duplex channels (i.e. bi-directional but not at the same time). This property is accomplished by means of a different mode of operation for the communicating matrix and the server. Section 6.2 is dedicated to studying the aspects related to the correctness of the implementation (specification) of the constructs for the transmission of messages proposed for the CSXMS-c. In particular, the results of [34] that are presented here, show how these constructs implement correctly the message-passing within the communicating states and why the handling by the components of the communicating matrix is correct.

Another approach to communicating EXM called the *modular specification of systems using (Eilenberg) communicating X-machines* (MSS) follows in section 7. The MSS methodology proposed by Kefalas *et al.* [35] is based on two steps:

1. The stand-alone EXM are specified independently of the communicating system and
2. Communication amongst components is determined.

The introduction to this model explains how it was originally based on some syntactical modifications of XMDL [18, 19] a descriptive language for EXM. We justify a different practice for specifying systems based on MSS, for the reason that our aim here is to provide reference information within the same or at least a similar standard mathematical notation to that that has been used in the body of the X-machine theory. With regard to this, we suggest to employ the *(Eilenberg) multiple-stream X-machine* model (M-SXM) presented in [27]. Section 7.1 explains how a communicating system of M-SXM can be constructed, by defining a subclass of M-SXM for which just one symbol is taken from any of the input streams and a symbol is produced for one of the output streams with each function that is executed. This model, called the *communicating (Eilenberg) multiple-stream X-machine* (CM-SXM), is defined here in order to accommodate the conditions of MSS. To be more specific, in [35] it is recognised that every time a machine performs a function just one input stream and one output stream are considered without regard to the number of them. Then we present a model for integrating these machines within a communicating structure.

The CM-SXMS can be seen as a graph G = (V, E) where the nodes are the CM-SXM, and the edges stand for streams that are shared between CM-SXM. Thus, the same stream is used as an output from one machine and as an input for another and therefore communication is possible. To maintain consistency with respect to the detect-ability of behaviours, in this model it is also assured that each component has one standard input stream and one

standard output stream, that is to say, streams that are not used for communication but for interacting with the environment. Section 7.2 explains how the farmer process can be specified by a CM-SXMS. Finally, section 8 summarises the conclusions of this report.

3. The communicating (Eilenberg) X-machines systems (CXM-system)

The *communicating (Eilenberg) X-machines* (CXM) are EXM with one or more input and output ports, where each output port of a machine is connected to an input port of another machine by a channel and a data item or a signal can be transmitted through this channel. The definitions of the next section are taken from [28].

3.1 The CXM-system model

Definition 1: A CXM is given by $\Lambda = (X, Q, \Phi, F, Pre, Ps, I, T)$ where:

- $X = \Gamma^* \times M \times \Sigma^*$
- $\Sigma^* = \prod_{j=1}^{m} \Sigma_j$ and $\Gamma^* = \prod_{i=1}^{t} \Gamma_i$,
  with Σ_j and Γ_i are the alphabets of the *j-th* input port and *i-th* output port respectively, and *m* and *t* are the numbers of input and output ports respectively.
- M is the data type of the CXM memory.
- Q is the finite set of states of the CXM.
- Φ is the set of relations on X:
  $\Phi : P(X \leftrightarrow X)$
- F is the next-state function that is often described by means of a state-transition diagram:
  $F : (Q \times (\Phi \times Pre)) \rightarrow Q$
- *Pre* is the set of *predicates* on $\Sigma^* \times M$, such that each predicate can be associated with one or more transition.
- *Ps* is the set of ports. Each port has a name, a classification (input or output port) and an associated alphabet.
- I and T are the sets of initial and final states:
  $I \subseteq Q, T \subseteq Q$

Definition 2: A CXM-system of *n* CXM is a pair $W_n = (R, E)$ where:

- $R = \{\Lambda_k \; \forall \; 1 \leq k \leq n\}$ is a set of *n* CXM and
- E is a set of relations $\prod_{i=1}^{t} \Gamma^*_{i,k} \leftrightarrow \prod_{j=1}^{m} \Sigma^*_{j,k'}$

It is clear that the channels link ports of different machines where each channel is represented as a relation between an output port and an input port and, as the terminology suggests, the communication between CXM is achieved via the channels.

A fundamental issue that was suggested in [28] is that there are two different operational aspects that need to be modelled by a system of communicating machines; the *external* and *internal behavioural models*. The first of these models is simply determined by the communication amongst components. The idea behind the latter refers to considering the internal behaviour of each component, and in this context, the set of states and transitions can be viewed as the behaviour of each one of them. This is of course an essential feature for any specification formalism for distributed systems which is to be useful in practice, since it allows each process (i.e. component) to be specified separately, and then the components to be combined at the end.

3.2 The input-output relationship for the CXM-systems

In this section, we show that the CXM-system model was not developed to the point of directly deriving the input-output relationship from it, and seems that the most natural way to investigate this question is by means of identifying the type of EXM that this model employs.

It appears that a CXM is just an EXM with ports. However, this is not the case since by definition an EXM reads a single input from the environment that is codified into the memory (by means of a coding function). Then from an initial control-state, a function that can process the memory is chosen and the machine moves to a new control-state and a new memory value is calculated. The machine continues in this way until there are no processing functions available and if the machine is in a final state then the last memory value is decoded to the environment (by means of a decoding function).

On the other hand, a CXM does not necessarily receive a single input nor produce a single output. The selection of a transition depends on a predicate, which is defined for the ports or, more precisely, on the input streams presented at them and on the current memory value. Therefore, a pair of these ports, let us say *Ix* and *Ox* can be considered respectively as the input and output ports from the environment. Thus, it is always possible to design a machine with transitions that read from or write to such ports (in a stream fashion) during its operation. If this is the case then the CXM behaves like an SXM.

To be more precise the CXM seems to behave like a *generalised* SXM (GSXM). Even though it is not explicitly defined, it is easy to see with the example presented in [28] that it is possible to specify a predicate for any port to evaluate whether that port is equal to the *nil* value (i.e. the port is empty). Consequently, a CXM can be designed in such a way that it reads a symbol from *Ix* in its first transition and produces an output to *Ox*. That machine can have a path of transitions that evaluates whether *Ix* is equal to *nil*, and if each of these transitions writes a symbol to *Ox* then we have a GSXM.

To justify the claim that a CXM can be treated as a Sm-SXM (see definition 49 in [27]), let us consider the situation in which a CXM has a transition that reads from *Ix* and writes to *Ox*, this corresponds to the set of *non-empty operations*. If the machine has a transition that reads from *Ix* and writes to any other port different from *Ox*, this corresponds to the set of *empty-output operations*. If it has a transition that reads from any port other than *Ix* (or may be the predicate corresponds to *Ix* but evaluates if such a port is empty) and writes to *Ox*, this corresponds to the set of *empty-input operations*. Finally, if the CXM has a transition that reads from and writes over to two ports other than *Ix* and *Ox* this corresponds to the set of *empty operations*.

Nevertheless, the assumptions here were artificial in the sense that it is assumed that a couple of hypothetical ports called *Ix* and *Ox* exist (i.e. there is a restricted and clear communication with the environment) but the CXM model does not have this restriction. In fact, each machine can have a number of ports (including zero) connected to the environment. From this, it appears that the specific nature of the CXM-system as defined in [28] needs to be extended if the derivation of the input-output relationship is required, which is of course the case if the SXMT testing methods is contemplate.

Assuming we have the *Ix* and *Ox* ports (and they are the only ports that interact with the environment) where the symbols arrive dynamically into *Ix* while the machine is operating. Then there is a need to know where this stream of symbols terminates. In this respect, it is interesting to note that for the *family of (Eilenberg) stream X-machines* (SXM, Sm-XM, GSXM) the form of the type indicates that any function is undefined when the input stream is empty. Hence if the machine reaches such a condition then it will stop (ideally in a termination state). Unfortunately, this situation is not contemplated in [28] where it is supposed that the alternating bit protocol presented there will be executed forever. To illustrate the difficulties associated with this, consider that a CXM is in a particular state (which may be a termination one). Let us say that there is only one function emerging from that state (this can be easily extended to many functions) that is triggered by the arrival of a certain symbol to *Ix*. The question is, should the machine stop since the port is empty, even if the symbol will get there later, or should it wait even if the end of the stream was reached. One way to solve such a situation is by associating a symbol to indicate the *end-of-stream* (*eos*) or alternatively by providing the length of the input stream in advance (e.g. the first symbol can indicate this).

3.3 Specifying a process farm with a CXM-system

This section is dedicated to the study of a complete and original example of a CXM-system specification. Let us consider the process farm approach, which is applicable to problems whose solution will decompose into smaller independent parts that can be executed concurrently and the effect summed at the end. A generic structure for the process farm with *k* workers is shown in figure 1, where following Barnard *et al.*, the processes are drawn as ovals, the input ports as circles and the output ports as rectangles.

In a process farm, the *farmer* process generates (or fetches) sub-problems that are usually represented by sets of data known as work packets (or tasks). The farm distributes the tasks to the *workers* through the *Packet* channels

(i.e. each new task is passed to any free worker) and subsequently each worker solves the individual sub-problem, produces a result, and sends it to the *reaper* using the corresponding *Result* channel. Every time a worker is free, it requests a new data packet from the farmer using its *Ready* channel. The reaper collects and integrates all the results, and then produces a solution.

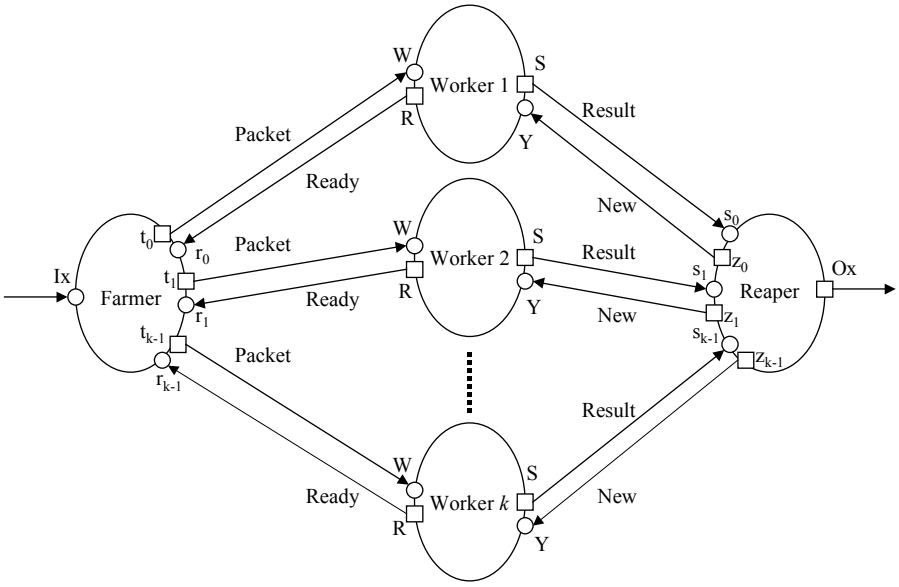


Figure 1

It is important to note that the nature of the predicates defined on Σ* x M together with the non-buffered ports imposes the use of extra connections. It is easy to see the need for this from the following illustration. Let t and t' be the times when a particular worker sends two consecutive results to the reaper whence $t' > t$ and let t_0 and t_f be the times when the reaper takes these results from that worker. If $t_0 < t$ and $t' < t_f$, because there is no buffer involved and since the predicates in the worker cannot check the state of the output ports then the first result produced by the worker will be lost.

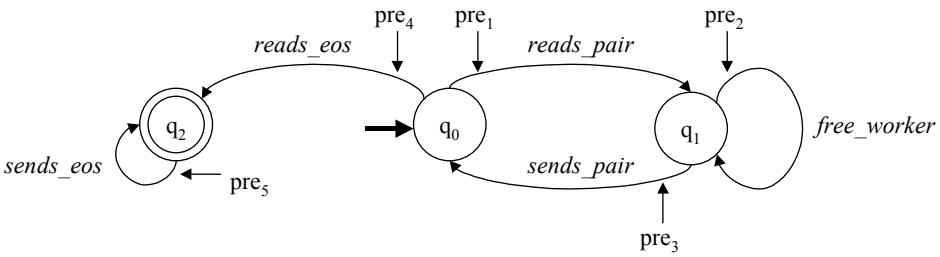
To solve the kind of synchronisation problem mentioned above, it is necessary to include another channel, which may not have been part of the original design, between the reaper and each one of the workers. The new channel is called *New* and it is used in a similar way to the Ready channel. In brief, when the reaper process has already taken the last packet from a particular worker, then it asks for a new data packet from the same worker using this channel. To complete the example, we utilise as in [27] the computation of the scalar product of two positive integer vectors of the same order $v_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and $v_2 = \langle b_1, b_2, \ldots, b_n \rangle$ where the scalar product is defined as:

$$\pi = v_1 \bullet v_2 = \sum_{i=1}^{n} (a_i \text{ x } b_i)$$

a. farmer process.

To specify the farmer process using a CXM, it is assumed that the farmer will fetch from its input port *Ix* the vectors' elements in pairs in the following order (a_1, b_1), (a_2, b_2) and so on (in *reads_pair*). Then the farmer will check all its r_j ports to find a free worker in a round-robin fashion (in *free_worker*). If no worker is free, this loop continues until a worker requests a new task. The farmer then sends the pair (a_i, b_i) to that worker (using t_j port in *sends_pair*) and fetches a new pair (again in *reads_pair*). This process continues until the *eos* symbol is received (in *reads_eos*). Before the farmer stops, it resends the *eos* message to all the workers, waiting in each case for the request of a new task (in *sends_eos*).

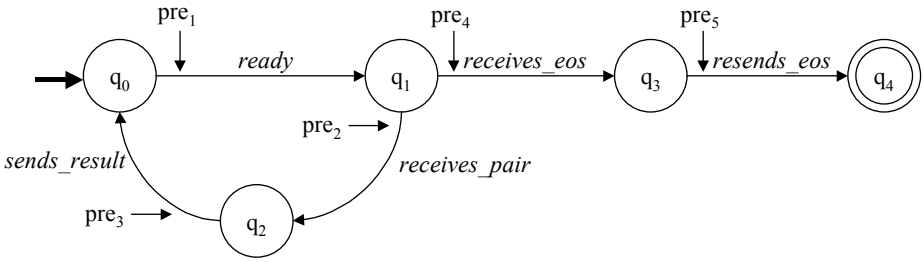
The state-transition diagram and the set of functions for the CXM corresponding to the farmer are shown below in figure 2. In the same way as the original notation proposed for this model, each transition is associated with a predicate, which is graphically represented as an incoming arrow.



pre_1: ($Ix \neq nil$) and ($Ix >= 0$):
reads_pair(<*Ix*, *r*>, *m* = <*k*, *p*, *w*, *e*>, *t*) = (<*Ix'* ← *nil*, *r*>, *m'* = <*k*, *p* ← *Ix*, *w*, *e*>, *t*)
pre_2: (*r*[*m.w*] = *nil*):
free_worker(<*Ix*, *r*>, *m* = <*k*, *p*, *w*, *e*>, *t*) = (<*Ix*, *r*>, *m'* = <*k*, *p*, *w'*← (*w* + 1) % *k*, *e*>, *t*)
pre_3: (*r*[*m.w*] ≠ *nil*):
sends_pair(<*Ix*, *r*>, *m*, *t*) = (<*Ix*, *r'*>, *m*, *t'*) where
∀ 0 ≤ *i* ≤ *m.k* – 1 | *i* ≠ *m.w* *r'*[*i*] = *r*[*i*] and *t'*[*i*] = *t*[*i*] and
r'[*m.w*] = *nil* and *t'*[*m.w*] = *m.p*
pre_4: (*Ix* ≠ *nil*) and (*Ix* = *eos*):
reads_eos(<*Ix*, *r*>, *m*, *t*) = (<*Ix'* ← *nil*, *r*>, *m*, *t'*)
pre_5: (*r*[*m.e*] ≠ *nil*) and (*m.e* < *m.k*):
sends_eos(<*Ix*, *r*>, *m* = <*k*, *p*, *w*, *e*>, *t*) = (<*Ix*, *r'*>, *m'* = <*k*, *p*, *w*, *e* ← *e* + 1>, *t'*) where
∀ 0 ≤ *i* ≤ *m.k* – 1 | *i* ≠ *m.e* *r'*[*i*] = *r*[*i*] and *t'*[*i*] = *t*[*i*] and
r'[*m.e*] = *nil* and *t'*[*m.e*] = *eos*

Figure 2

For the sake of simplicity, the memory of the farmer is written as 4-tuple of the form *m* = <*k*, *p*, *w*, *e*> where *m.k* is an integer that corresponds to the number of workers in the system. *m.p* is used to store the last pair of elements received. *m.w* keeps track of the number of the port of the last worker that has received a task, and from *m.w* plus one modulo *m.k*, the loop restarts when the machine looks again for a free worker. Finally, *m.e* is used to control the loop for sending the *eos* symbol to all the workers. The initial memory is thus m_0 = <*k*, (0, 0), 0, 0>. Additionally, when it is necessary to refer to a particular memory element, let us say *p*, the dot notation just described is employed. The set of input ports is a pair <*Ix*, *r*> where *Ix* is as was explained above and *r* is an array where each *r*[*i*] is the port connected to the Ready channel that goes from $worker_{i+1}$ to the farmer (i.e. *r*[*i*] is r_i in figure 1). The set of output ports is modelled in a similar way in this case using the array *t*, where *t*[*i*] is the port connected to the Packet channel. It should be noted that the ports range from *t*[0] to *t*[*k* - 1] and from *r*[0] to *r*[*k* - 1] and the workers range from $worker_1$ to $worker_k$.



pre_1: (*W* = *nil*):
ready(<*W*, *Y*>, *m*, <*R*, *S*>) = (<*W*, *Y*>, *m*, <*R'* ← *ok*, *S*>)
pre_2: (*W* ≠ *nil*) and (*W* ≠ *eos*):
receives_pair(<*W*, *Y*>, *m*, <*R*, *S*>) = (<*W'* ← *nil*, *Y*>, *m'* ← *W.a* * *W.b*, <*R*, *S*>)
pre_3: (*Y* ≠ *nil*) and (*Y* = *ok*):
sends_result(<*W*, *Y*>, *m*, <*R*, *S*>) = (<*W*, *Y'* ← *nil*>, *m*, <*R* , *S'* ← *m*>)
pre_4: (*W* ≠ *nil*) and (*W* = *eos*):
receives_eos(<*W*, *Y*>, *m*, <*R*, *S*>) = (<*W'* ← *nil*, *Y*>, *m*, <*R*, *S*>)
pre_5: (*Y* ≠ *nil*) and (*Y* = *ok*):
resends_eos(<*W*, *Y*>, *m*, <*R*, *S*>) = (<*W*, *Y'* ← *nil*>, *m*, <*R*, *S'* ← *eos*>)

Figure 3

b. worker process.

The general layout of the workers is illustrated in figure 3. For this case, we have two pair of ports <*W*, *Y*> and <*R*, *S*> that connect each worker with the farmer and the reaper as in figure 1.

At the beginning, the workers send a message through their corresponding Ready channel to the farmer (using the port *R* in *ready*) to indicate that they are free. The predicate for this is that the port *W* is empty. After that, each worker waits until a new message arrives into the port *W* and when this happens there are two possibilities:

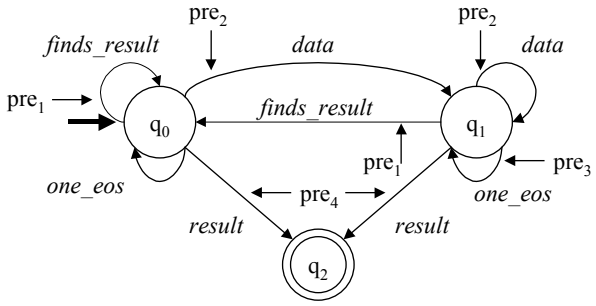
1. If the message is the *eos* symbol (in *receives_eos*) then the worker resends such a symbol to the reaper using the Result channel when it is ready to receive (in *resends_eos*). Afterwards the worker stops in terminal state q_4.
2. If the message is not the *eos* symbol then the worker takes the message that corresponds to the pair (a_i, b_i) and multiplies $a_i * b_i$ (in *receives_pair*). After that, the worker waits to receive an acknowledgement from the reaper. When this arrives, the result is sent to the reaper (in *sends_result*) and the process starts again from state q_0.

c. reaper process.

Clearly, the reaper uses a loop (as the farmer process does) to determine whether a worker has already sent a result (in *finds_result*). If so, the reaper receives and adds it up to the memory accumulator (in *data*). Otherwise, the loop *finds_result* continues sending a message to the workers one by one to indicate that it is ready to receive, until a worker transmits a new result.

When a worker receives the *eos* symbol then it resends this symbol to the reaper. At this moment, the reaper performs the *one_eos* operation which decrements the memory variable that contains the number of "active" workers. Eventually there will not be any worker active. Because of this, the reaper will transmit the memory value that contains the addition and then it will stop (in *result*).

The computation of the reaper process is illustrated in figure 4, where obviously a state-transition diagram with two states can be drawn (i.e. q_0 and q_1 can be joined in one state), nevertheless, we decide to present a three state machine here to produce a clearer and more understandable representation.



pre_1: ($s[m.w] = nil$) and ($m.t > 0$):
finds_result(s, m = <k, p, w, t>, <z, Ox>) = (s, m' = <k, p, (w + 1) % k, t>, <z', Ox>)
where $\forall\ 0 \le i \le\ m.k - 1 \mid i \neq m.w\ \ z'[i] = z[i]$ and $z'[m.w] = ok$

pre_2: ($s[m.w] \neq nil$) and ($s[m.w] \neq eos$) and ($m.t > 0$)
data(s, m = <k, p, w, t>, <z, Ox>) = (s, m' = <k, $p + s[m.w]$, (w + 1) % k, t>, <z', Ox>)
where $\forall\ 0 \le i \le\ m.k - 1 \mid i \neq m.w\ \ s'[i] = s[i], z'[i] = z[i], s'[m.w] = nil$
and $z'[m.w] = ok$

pre_3: ($s[m.w] \neq nil$) and ($s[m.w] = eos$) and ($m.t > 0$)
one_eos(s, m = <k, p, w, t>, <z, Ox>) = (s', m' = <k, p, (w + 1) % k, t - 1>, <z, Ox>)
where $\forall\ 0 \le i \le\ m.k - 1 \mid i \neq m.w\ \ s'[i] = SX[i]$ and $s'[m.w] = nil$

pre_4: ($m.t = 0$)
result(s, m, <z, Ox>) = (s, m, <z, $Ox' \leftarrow p$>)

Figure 4

The reaper memory is organised as a 4-tuple <k, p, w, t> of integers where $m.k$ is the number of workers in the system. $m.p$ is used as an accumulator of the partial results. $m.w$ is used as a port index in a similar way as the farmer uses it, and $m.t$ contains the number of active workers in the system. The initial memory is m_0 = <k, 0, 0,

$t>$ with $m.k = m.t$. Each input port is an element of the array s where $s[i]$ corresponds to the port that connects the $worker_{i+1}$ to the reaper. The set of output ports is a pair $<z, Ox>$ where Ox as has already been described and z is an array of ports such that $z[i]$ corresponds to the port that connects the New channel to the $worker_{i+1}$.

4. The communicating (Eilenberg) stream X-machines systems (CSXMS)

The CSXMS is a relatively new specification formalism (1999) that has been developed and presented for specifying distributed systems. The CSXMS combines the advantages of the EXM (i.e. to model both the system and data control structure while allowing them to be specified separately) with the possibility of sending and receiving messages among several of these abstract machines. The definitions of the following section are taken and adapted from [31, 33, 34].

4.1 The CSXMS model

Definition 3: A CSXMS with n components is a triplet $W_n = (R, CM, C^0)$ where:

- R is the set of $n = |R|$ components of the system of the form $P_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0)\ \forall\ 1 \leq i \leq n$. From now on such components will be called *communicating machines*. Λ_i is an SXM with memory M_i (for further details see definition 5). IN_i and OUT_i are the respective input and output ports of the *i-th* communicating machine, with the property that $IN_i, OUT_i \subseteq M_i \cup \{\lambda\}$ and $\lambda \notin M_i$. The symbol λ is used to indicate an empty port. The initial port values are in_i^0 and out_i^0.
- CM is the set of matrices of order n x n which form the values of a matrix variable that is used for communication between the machines, so for any $C \in CM$ and any pair i, j we have that the value in $C[i, j]$ is a message from the machine $P_i \in R$ to the machine $P_j \in R$. Therefore, each element $C[i, j]$ can be thought of as a variable that is used as a temporary buffer and $IN_i \subseteq C[i, j] \subseteq OUT_j$.
- All the messages passing between communicating machines P_i and P_j will be values from M_i and M_j (i.e. the memory of Λ_i and Λ_j). The symbol λ in the matrices is used to indicate that there is no message. The symbol @ is used to indicate a channel that is not going to be used (e.g. from a communicating machine to itself), so that there is no possibility of sending a message with this value. The elements of the matrices are therefore drawn from $M \cup \{\lambda, @\}$ where:

$$M = \bigcup_{i=1}^{w} M_i \text{ and } \lambda, @ \notin M.$$

- C^0 is the initial communication matrix where it is defined that $C^0[i, j] = \lambda$ if communication is allowed between machines P_i and P_j, otherwise $C^0[i, j] = @$ (e.g. $C^0[i, i] = @$).
- The *i-th* communicating machine can only read from the *i-th* column and write to *i-th* row of the matrix (see definition 5 for a detailed explanation of communicating functions).

Definition 4: For any $C \in CM$, any value $v \in M$ and any pair of indices $1 \leq i, j \leq n$, with $i \neq j$.

- If $C[i, j] = \lambda$ an *output variant* of C, denoted by $C_{ij} \Leftarrow v$ is defined as:
  $(C_{ij} \Leftarrow v)[i, j] = v$ and $(C_{ij} \Leftarrow v)[k, m] = C[k, m]\ \forall\ (k, m) \neq (i, j)$
- If $C[i, j] = v$ an *input variant* of C, denoted by $\Leftarrow C_{ij}$ is defined as:
  $(\Leftarrow C_{ij})[i, j] = \lambda$ and $(\Leftarrow C_{ij})[k, m] = C[k, m]\ \forall\ (k, m) \neq (i, j)$

These variants effectively define the allowable transitions from one matrix to another. The concept of variants is fundamental to the concept of configuration.

Definition 5: A *communicating machine* is a 5-tuple $P = (\Lambda, IN, OUT, in^0, out^0)$ where:

$\Lambda = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ is a SXM with the following properties:

- Σ and Γ are finite sets respectively called the input and output alphabet.
- The set of finite states Q has to be partitioned as $Q = Q' \cup Q''$ where Q' is called the set of *processing states* and Q" is the set of *communicating states* and $Q' \cap Q'' = \varnothing$. If $q' \in Q'$ then all the functions emerging from q' are *processing functions*. If in q' several functions can be applied, then one of them is arbitrarily chosen otherwise (i.e. if no function can be applied) the communicating machine and the entire system block. If $q'' \in Q''$ then all the functions emerging from q'' are *communicating functions*. If

in *q"* several functions can be applied, then one of them is arbitrarily chosen, if not then the machine does not change the state and waits until one function can be applied.

- M is a (possibly infinite) set called the memory.
- The type of the machine is the set $\Phi = \Phi' \cup \Phi''$ where Φ' is called the set of *processing functions* or *ordinary functions* and Φ'' is the set of *communicating functions* and $\Phi' \cap \Phi'' = \varnothing$. The elements of Φ' are relations (partial functions) of the form:

  Φ': IN x M x OUT x $\Sigma \rightarrow \Gamma$ x IN x M x OUT

  (I) A processing function ϕ' acts as in an ordinary SXM that is

  $\forall x \in$ IN, $\forall m \in$ M, $\forall y \in$ OUT

  $\phi'(x, m, y, <>) = \perp$

  $\forall x \in$ IN, $\forall m \in$ M, $\forall y \in$ OUT, $\forall h \in \Sigma$, $\forall s^* \in \Sigma^*$, $\forall g^* \in \Gamma^*$

  if $\exists m' \in$ M, $t \in \Gamma$, $x' \in$ IN, $y' \in$ OUT that depend on m, h and x then

  $\phi'(x, m, y, h::s^*) = (g^*::t, x', m', y')$

  otherwise

  $\phi'(x, m, y, h::s^*) = \perp$

  (II) A communicating function ϕ'': Φ'': IN x OUT x *CM* $\rightarrow$ IN x OUT x *CM* can be one of the following forms:

  (II.A) An *output-move*. This form is used by machine P_i to send a message to machine P_j, using $C[i, j]$ as a buffer. The set of moves from the output port of P_i to $C[i, j]$ are called *output moves* and are denoted by $MVO_i \; \forall \; 1 \leq i \leq n$. Thus $MVO_i = \{mvo_{i->j} \mid 1 \leq j \leq n, i \neq j\}$ where $mvo_{i->j}$: OUT_i x *CM* $\rightarrow$ OUT_i x *CM* is defined by:

  $\forall y \in OUT_i$, $\forall C \in CM$

  if $\exists j \mid j \neq i$ and $y \neq \lambda$ and $C[i, j] = \lambda$ (i.e. y is not empty and $C[i, j]$ is empty)

  $mvo_{i->j}(y, C) = (y \leftarrow \lambda, (C_{ij} \Leftarrow y))$ (i.e. the result is an output variant of C_{ij})

  (II.B) An *input-move*. This form is used by machine P_i to receive a message from machine P_j using $C[j, i]$ as a buffer. The set of moves from $C[j, i]$ to the input port of P_i are called *input moves* and are denoted by $MVI_i \; \forall \; 1 \leq i \leq n$. Thus $MVI_i = \{mvi_{j->i} \mid 1 \leq i \leq n, i \neq j\}$ where $mvi_{j->i}$: IN_i x *CM* $\rightarrow$ IN_i x *CM* is defined by:

  $\forall C \in CM$

  if $\exists j \mid j \neq i$ and $x = \lambda$ and $C[j, i] \neq \lambda$ (i.e. the input port is empty and $C[i, j]$ is not)

  $mvi_{j->i}(\lambda, C) = (x \leftarrow C[j, i], \; (\Leftarrow C_{ji}))$ (i.e. the result is an input variant of C_{ij})

  That is $mvo_{i->j}$ can be executed only when $C[i, j]$ is empty and $mvi_{j->i}$ can be executed only when $C[j, i]$ is not empty. Additionally, it is assumed that all the operations concerning the same cell of the communication matrix are done under *mutual exclusion*.

- The set of communicating functions Φ'' is made up as:

  $\Phi'' \subseteq MVO_i \cup MVI_i \; \forall \; 1 \leq i \leq n$.

- All the relations (partial functions) of $\Phi = \Phi' \cup \Phi''$ are extended to:

  Φ_E: IN x M x OUT x *CM* x $\Sigma \rightarrow \Gamma$ x IN x M x OUT x *CM*

  All the extended relations (partial functions) will operate only on the variables for which the relation (partial function) is defined leaving the rest unchanged. For uniform treatment, the same notation will be used for what follows (we will use Φ instead of Φ_E) where the relations (partial functions) are distinguished by their domains.

- The next-state function is:

  F : Q x $\Phi \rightarrow P$(Q) with *domain*(F) $\subseteq$ (Q' x Φ') $\cup$ (Q" x Φ'')

- I $\subseteq$ Q and T $\subseteq$ Q are respectively the sets of initial and terminal states and $m_0 \in$ M is the initial memory value.

4.2 Specifying a process farm with a CSXMS

In this section, we shall restructure the process farm example assuming as in section 3.3 that the input to the system is given by the vectors $v_1 = <a_1, a_2, \ldots, a_n>$ and $v_2 = <b_1, b_2, \ldots, b_n>$ where each symbol in the farmer's input stream is a pair (a_i, b_i). The system's organisation is presented in figure 5.

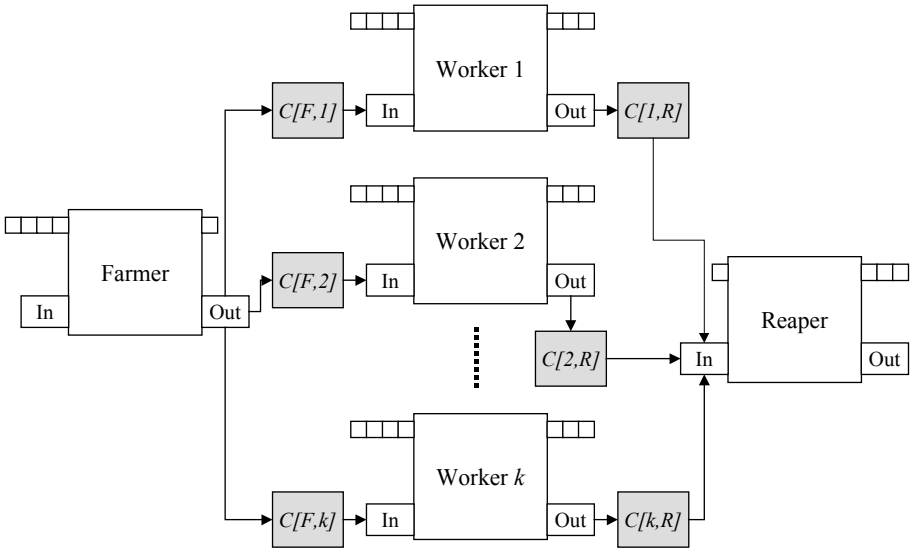
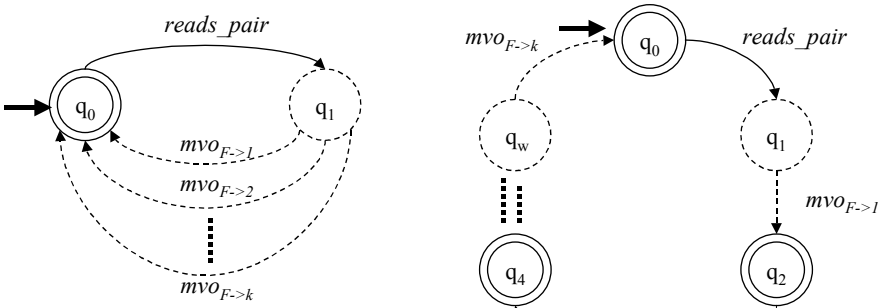


Figure 5

The arrangement of the process farm with k workers can be as follows; let us say that the components P_1, P_2,…, P_k model the workers and that the components P_F and P_R correspond respectively to the farmer and the reaper where $F = k + 1$ and $R = k + 2$. From figure 5, it is clear that fewer connections (i.e. channels or matrix cells) are needed than in the CXM-system. To be more specific, in the CXM-system with k workers $4k$ channels are used whilst here $2k$ cells are required, the reason for this is that the synchronisation between any two processes will be supported by the model operation itself.

a. farmer process.



(a) non-deterministic

(b) deterministic

$reads\_pair(x, m, y, C, h) = (h, \lambda, m, y' \leftarrow h, C)$
$mvo_{F->1}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y \leftarrow \lambda, (C_{F1} \Leftarrow y))$
$mvo_{F->2}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y \leftarrow \lambda, (C_{F2} \Leftarrow y))$

$mvo_{F->k}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y \leftarrow \lambda, (C_{Fk} \Leftarrow y))$

Figure 6

The farmer can be specified either in a non-deterministic manner (figure 6.a) or deterministically (figure 6.b). In both cases, the process starts its operation in state q_0 where a pair (a_i, b_i) is read and removed from the input stream whilst it is added to the output stream. The output port is also updated with the same pair and the machine

moves to state q_1 (in *reads_pair*).

For the non-deterministic specification, from state q_1 there are k communicating functions emerging from it back to q_0. Specifically these functions are output-moves of the form $mvo_{F->K}$ where F is the index in the communication matrix of the farmer and K ranges from 1 to k (i.e. one for each worker). Now, because q_1 is a communicating state, if no function can be applied then the process waits until one function can be executed, on the other hand if several functions can be computed one of them is arbitrarily selected. In brief, if several workers are free one of them is chosen arbitrarily.

For the deterministic specification, once the machine is in state q_1 there is just one possibility of communication (in function $mvo_{F->1}$). So the machine will be able to continue its operation only when the $worker_1$ takes the previous packet sent to it (i.e. when the cell $C[F, 1]$ is empty) and when the function $mvo_{F->1}$ is applied this data is transmitted to the $worker_1$. The procedure continues in the same way from state q_2 where *reads_pair* is executed again, followed by $mvo_{F->2}$ and then a packet will be sent to $worker_2$ and so on.

It is important to mention that this specification is similar to the specification using a CXM-system (section 3.3) in the sense that in both of them the selection of a free worker is done by trying one after the other in order (i.e. round-robin). Nevertheless, in the CXM-system when a worker is not ready to receive, the farmer tries with the next one (see *free_worker*) while here for the same case the farmer waits until the worker is ready.

On one hand, the CXM-system specification can be easily modified to behave in the same way as the CSXMS specification by eliminating the increment operation of memory value w from function *free_worker* and by including it in *read_pairs*. In this way, the farmer would try to send a message to the same worker repeatedly and it will proceed with the next worker only after a new pair has been fetched (see figure 2). On the other hand, the deterministic specification of the farmer in this section could also be modified to operate as the CXM-system specification although this will imply the use of more connections. In other words, every cell $C[K, F]$ would be needed to indicate to the farmer that the $worker_k$ is ready (as the Ready channel is used in section 3.3). Since our intention in this example is to illustrate among other things how the CSXMS model can be used to achieve synchronisation. We keep the specification within the idea that the ready acknowledgment is given by a worker when it takes the data from the corresponding communicating cell, as explained below.

In any of the cases (deterministic or non-deterministic), the task that is modelled with this corresponds to finding a free worker and the synchronisation between the farmer and the workers is supported by the CSXMS. To be more precise, an output-move requires the communication cell to be empty (i.e. $C[F, K] = \lambda$), once the cell has a value it acts as a temporary buffer and the cell is left empty after it has been read. These conditions ensure that:

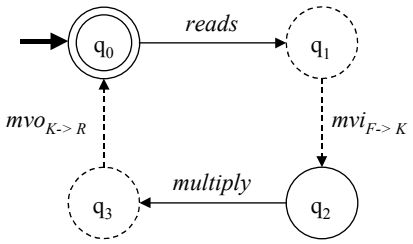
1. The farmer does not send a packet to a worker until it has explicitly indicated in some way that is ready to receive it (i.e. by emptying the corresponding cell) and
2. The way that a worker requests a new data packet is by taking the packet from the cell, leaving it empty. This works even in the case when the worker is performing some computation because if a new data packet is put into the cell, this guarantees that the current operation is not interrupted (i.e. the cell acts as a buffer).

These two invariants provide an equivalent mechanism for exchanging messages between the farmer and the workers as the one described for the CXM-system where the channels Packet and Ready are needed to achieve the same effect.

b. worker process.

We shall include the input and output streams for the workers even when the data packets and the results that they need come and go respectively from the farmer or to the reaper. If these streams were not included then the specification for the workers would be a Sm-SXM instead of a SXM. Consequently, the specification for each worker P_i should have in its input stream the unary representation of a even number, let us say u_i (i.e. a particular symbol is repeated u_i times) such that $u_1 + u_2 + \ldots + u_k = 2n$ whence n is the order of the vectors. What this number u_i represents is twice the number of operation that P_i is going to perform. The reason for doing this, it is to ensure that the n pairs will be processed by the whole system and that every worker's processing function has a symbol to remove from the stream. The number u_i is even, because two operations are needed for the concluding one computation (i.e. loop). The first is for the reception of a pair from the farmer, and the second has to do with the calculation and the transmission of the result. Additionally, this number will indicate the workload that a worker has to do, to equilibrate this workload among all the workers the value u_i has to be

approximated 2*n* div *k*.A formal specification for the worker process is now possible, and this is presented in figure 7.

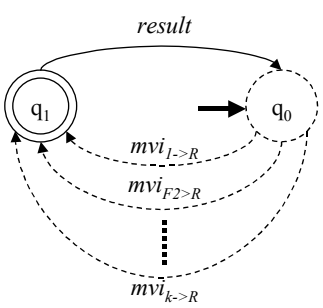


$reads(x, m, y, C, h) = (h, \lambda, m, y, C)$
$mvi_{F->K}(\lambda, m, y, C\ ,\ \varepsilon) = (\varepsilon, x \leftarrow C[j, i], m, y, (\Leftarrow C_{ji}))$
$multiply(x,, m, y, C, h) = (h, x, m, y' \leftarrow x.a * x.b, C)$
$mvo_{K->R}(x, m, y, C, \varepsilon) = (\varepsilon, x, m,\ y\ \leftarrow \lambda, (C_{KR} \Leftarrow y))$

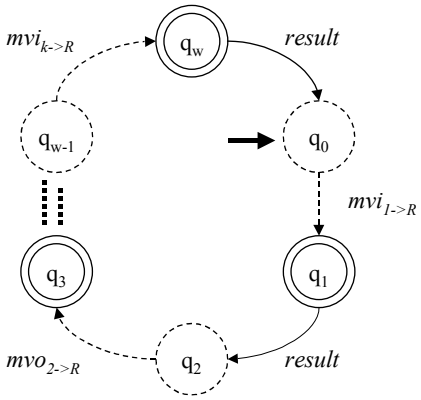
Figure 7

At the beginning that is from state q_0 the function *reads* is executed, moving the control to communication state q_1 and leaving the input port empty which is a necessary precondition for the following input-move. Then from q_1 the function $mvi_{F->K}$ is executed when possible (i.e. when $C[F, K] \neq \lambda$) where F and K are respectively the indexes of the farmer and the worker in the communication matrix. After that, the value in the input port is a pair (a_i, b_i) then function *multiply* computes $a_i * b_i$ and the result is left in the output port. Finally, the output-move $mvo_{K->R}$ sends the last result to the reaper where R is the matrix index for the reaper.

c. reaper process.



(a) non-deterministic



(b) deterministic

$mvi_{1->R}(\lambda, m, y, C\ ,\ \varepsilon) = (\varepsilon, x \leftarrow C[1, R], m, y, (\Leftarrow C_{iR}))$
$mvi_{2->R}(\lambda, m, y, C\ ,\ \varepsilon) = (\varepsilon, x \leftarrow C[2, R], m, y, (\Leftarrow C_{2R}))$

$mvi_{k->R}(\lambda, m, y, C\ ,\ \varepsilon) = (\varepsilon, x \leftarrow C[k, R], m, y, (\Leftarrow C_{kR}))$
$results(x, m, y, C, h) = (m', x, m' \leftarrow m + x, y, C)$

Figure 8

For the reaper specification, following a similar reasoning, as above, a particular symbol appears repeatedly one time after the other in the input stream, in such a way that such sequence codifies the unary representation of the order of one of the input vectors. As in the farmer there are two alternatives for specifying, that is deterministic (figure 8.a) or non-deterministic (figure 8.b).

Both specifications (deterministic and non-deterministic) operate under the same principles as the corresponding specifications for the farmer. The basic idea is that each function $mvi_{K->R}$ refers to a possible communication between the worker K and the reaper R and the function *results* adds to the memory the value of the input port. Thus, the memory is used as an accumulator and contains in any moment the partial summation of the scalar product of v_1 and v_2 where initially $m_0 = 0$. The result of this partial summation is concatenated in the output stream in all iterations. Therefore, the final symbol of the stream will be the value that corresponds to the scalar product of the two vectors.

We shall analyse the following situation for the non-deterministic case; let us suppose that worker_i has already sent a result to the reaper by means of its output-move, thus $C[i, R]$ contains a result. Now, if the reaper in state q_0 chooses a different input-move $mvi_{j->R}$ with $i \neq j$ this does not implies that the message from worker_i is lost because the cells of the communication matrix act as buffers and it is not hard to see that the number of symbols in the input stream of the reaper ensures enough repetitions of the whole operation of the machine in such a way that the message in $C[i, R]$ will be eventually received.

To conclude this section is important to remark that to ensure the correct operation of the whole system the specifications of the farmer and the reaper must be both deterministic or both non-deterministic. Additionally, if the specification is deterministic, the input streams of the workers have to be as follows. If $a = n \% k$ and $b = n$ div k then for all $1 \leq i \leq a$, $u_i = 2b + 2$ and for all $a < j \leq k$, $u_j = 2b$. The reason for this is because both the farmer and the reaper interact with the workers in order from 1 to k for the transmission and reception of messages.

4.3 Changes of configuration and the relation computed by a CSXMS

To avoid confusion between the states and transitions of a single communicating machine and the states and transitions of the CSXMS (i.e. global states and global transitions), henceforth, the following conventions are used. The term *configuration* is utilised for the set of all possible global states of the system. The term *state* is used for each component (i.e. for each communicating machine) to refer to the combination of its control-state and memory values. The term *machine-state* is used for the control-state of the SXM embedded in the components. The term *transition* is used to refer to the allowable global transitions. The term *event* is used for the change of state inside a component. The terms *next-state function* and *type-function* are utilised for SXM. According to [31] we have the following two definitions.

Definition 6: The *state* of a component $P = (\Lambda, \text{IN}, \text{OUT}, \text{in}^0, \text{out}^0) \in R$ of a CSXMS has the form $cf = (m, q, x, y, s, g)$ where:

- $m \in \text{M}$ is the value of the memory of Λ.
- $q \in \text{Q}$ is the machine-state of Λ.
- $x \in \text{IN}$ and $y \in \text{OUT}$ are the values of the ports of P.
- $s \in \Sigma^*$ is the input sequence of Λ.
- $g \in \Gamma^*$ is the output sequence of Λ.

Definition 7: A *configuration* of a CSXMS $\text{W}_\text{n} = (R, CM, C^0)$ has the form $CF = (cf_1, cf_2, \ldots, cf_n, C)$ where $\forall\ 1 \leq i \leq n$, $cf_i = (m_i, q_i, x_i, y_i, s_i, g_i)$ is the state of the *i-th* component and C is the current communication matrix of the system. A configuration is initial if and only if for all cf_i, $m_i = m_{i0}$, $q_i \in \text{I}_\text{i}$ and $C = C^0$. A configuration is final if and only if for all cf_i, $\text{s}_\text{i} = <>$ and $q_i \in \text{T}_\text{i}$.

To explain informally, a configuration of a CSXMS consists of the states of all the components together with the collection of messages in transit. Initial configurations are those in which each component is in an initial machine-state and the communication matrix is empty. A configuration is final if the input stream is empty and the machine-state is a final one for each component. In order to facilitate further analysis of the changes of configuration, we shall introduce the concepts of single change of configuration, communication steps, and processing steps as follows:

Definition 8: A configuration $CF' = (cf'_1, cf'_2, ..., cf'_n, C')$ is *reachable by one step* from another configuration $CF = (cf_1, cf_2, ..., cf_n, C)$ denoted by $CF \models^C CF'$, if $\exists\ i, j$ such that $\forall\ 1 \leq k \leq n$ with $k \neq i$, $cf_k = cf'_k$ and for $cf_i = (m, q, x, y, s, g)$ either:

- $q' \in F_i(q, mvo_{ij})$ with $y \neq \lambda$ and $C[i, j] = \lambda$, where $mvo_{ij}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{i,j} \Leftarrow y) \in MVO_i$, thus $cf'_i = (m, q', x, y' \leftarrow \lambda, s, g)$ and $C'[i, j] = y$, this step corresponds to component P_i sending a message, and is denoted by $CF \models^S CF'$ or
- $q' \in F_i(q, mvi_{ji})$ with $x = \lambda$ and $C[j, i] \neq \lambda$, where $mvi_{ji}(x, m, y, C, \varepsilon) = (\varepsilon, C[j, i], m, y, \Leftarrow C_{j,i}) \in MVO_i$, thus $cf'_i = (m, q', x' \leftarrow C[j, i], y, s, g)$ and $C'[j, i] = \lambda$, this step corresponds to component P_i receiving a message, and is denoted by $CF \models^R CF'$.

Since the relations $\models^S$ and $\models^R$ correspond to transitions in the CSXMS that are associated with internal send/receive events, $CF \models^C CF' \Leftrightarrow (CF \models^S CF' \vee CF \models^R CF')$.

Thus, the notion of a communicating step is interpreted as two alternative operations that correspond to a send or receive event (i.e. internal event) and so they define a computation in which a machine moves from machine-state to machine-state as long as a message is communicated.

Definition 9: A configuration $CF' = (cf'_1, cf'_2, ..., cf'_n, C')$ is *reachable by one processing step* from another configuration $CF = (cf_1, cf_2, ..., cf_n, C)$, denoted by $CF \models^P CF'$, if $\exists\ i$ such that $\forall\ 1 \leq k \leq n$ with $k \neq i$, $cf_k = cf'_k$ and given $cf_i = (m, q, x, y, s, g)$ and $cf'_i = (m', q', x', y', s', g')$ we have that $q' \in F_i(q, \phi')$, $\phi'(x, m, y, C, h) = (t, x', m', y', C) \in \Phi'$ and $s = h::s'$ and $g' = g::t$.

Intuitively, a configuration CF' is reachable by one processing step from another configuration CF, if the former results from the latter by the application of exactly one processing function to one of the components of the system. The change of state of the component then produces a change in the configuration of the whole system.

Definition 10: A *single change of configuration* of a CSXMS is defined as $CF \models^1 CF' \Leftrightarrow (CF \models^C CF' \vee CF \models^P CF')$. The reflexive and transitive closure of $\models^C$, $\models^P$ and $\models^1$ are respectively denoted by $\models^{*C}$, $\models^{*P}$ and $\models^{*1}$.

The concepts of communication steps and processing steps are actually described in terms of changes of configuration because those internal events trigger changes in the global state of the system. Put differently, the transitions that correspond to a single change of configuration are produced by internal events of the components and such events are generated by the next-state function together with a type-function of the SMX. This can be thought of as small changes in the embedded SMX causing large changes in the system in a reaction that goes from the SXM to the communicating machines (i.e. components) and from them to the CSXMS. With reference to the internal events it is also important to note that they do not only affect the state of the component (i.e. $\models^P$), but can also affect and be affected by the communication matrix (i.e. $\models^C$).

The change of configuration concept has the following intuitive meaning in the literature [31, 34]. Let CF_0 be the initial configuration when the system initiates its execution, from that point in time, all the components may start their executions concurrently changing from state to state whilst they are almost certainly sending and receiving messages. This is the reason why a configuration is defined as the Cartesian product of the local states of the components and the communication matrix in a given instant. Nevertheless, the events can occur at different times and with different duration. As a result, a global observer who has access to the whole system can note that some components may be in specific states while others are in the course of a transition from one state to another (i.e. executing a function).

The concept of single change of configuration introduced here, permits reasoning about the system in terms of single events, that is in terms of changes of state of the individual components, taking into consideration just one of them per step. However, the concept of change of configuration as it has been defined in the context of CSXMS [31, 33] (see definition 11), is broader than a single change of configuration in the sense that it allows several single changes of configuration in one step. In this direction, it becomes clear that every single change of configuration is a change of configuration, but in the opposite direction, a change of configuration can be composed of several single changes that can be executed concurrently.

Definition 11: A *change of configuration* $CF = (cf_1, cf_2, ..., cf_n, C) \models CF' = (cf'_1, cf'_2, ..., cf'_n, C')$ where:

$\forall\ 1 \leq i \leq n$, $cf_i = (m_i, q_i, x_i, y_i, s_i, g_i)$, $cf'_i = (m'_i, q'_i, x'_i, y'_i, s'_i, g'_i)$ it is possible if $CF \neq CF'$ and

$\exists\ c_0, c_1, \ldots, c_n \in CM$ with $c_0 = C$ and $c_n = C'$ such that $\forall\ 1 \le i \le n$ either:

- $cf_i = cf'_i$ and $c_i = c_{i-1}$ or
- $\exists\ \phi_i \in \Phi_i$ such that $q_i' \in F_i(q_i, \phi_i)$ and $\phi_i(x_i, m_i, y_i, c_{i-1}, h_i) = (t_i, x'_i, m'_i, y'_i, c_i)$ where $s_i = h_i::s'_i$, $g'_i = g_i::t_i$, $h_i \in \Sigma_i \cup \{\varepsilon\}$ and $t_i \in \Gamma_i \cup \{\varepsilon\}$.

The reflexive and transitive closure of $\models$ is denoted by $\models^*$.

In order to establish what a configuration *CF(t)* is in time *t*, let us assume that a change of state in a component occurs atomically at the end of each function, and such a new state is the current state of the component until the next function terminates. In the same way, it is assumed that the matrix is modified (when required) at the end of the corresponding function.

Example 1: In figure 9 a space-time diagram for a CSXMS with three components is presented. At the beginning (i.e. in time t_0), the configuration is $CF(t_0) = (cf1_0, cf2_0, cf3_0, C^0)$. Let $t > t_0$ be another moment, as is shown in the diagram by the vertical two-headed arrow. Because P_1 and P_2 are respectively executing $F1_0$ and $F2_o$ their states are still $cf1_0$ and $cf2_0$. On the other hand, P_3 is in state $cf3_1$ after the execution of $F3_0$ and thus a message has already been sent to the communication matrix, which is now C^1. The configuration in time instant *t* is therefore $CF(t) = (cf1_0, cf2_0, cf3_1, C^1)$. In the same figure some other changes of configuration are illustrated apart from $CF(t_0) = (cf1_0, cf2_0, cf3_0, C^0) \models CF(t_1) = (cf1_0, cf2_0, cf3_1, C^1)$, which appears as the first circle from left to right over the line of configurations *CF*.

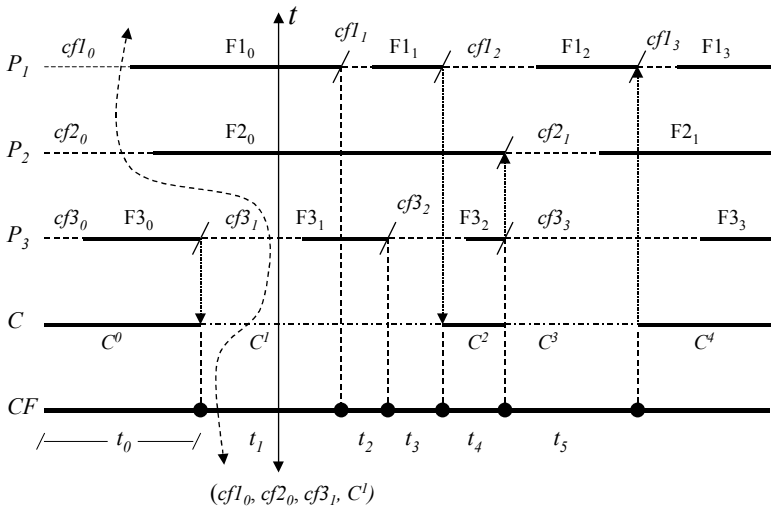


Figure 9

It must be noticed that the circles denote the time when a change of configuration occurs; nevertheless, the time references (i.e. such as t_0) correspond to intervals, that is they represent any time between the changes. Indeed, the changes of configuration that correspond to the circles in the figure are:

$CF(t_1) = (cf1_0, cf2_0, cf3_1, C^1) \models CF(t_2) = (cf1_1, cf2_0, cf3_1, C^1)$
$CF(t_2) = (cf1_1, cf2_0, cf3_1, C^1) \models CF(t_3) = (cf1_1, cf2_0, cf3_2, C^1)$
$CF(t_3) = (cf1_1, cf2_0, cf3_2, C^1) \models CF(t_4) = (cf1_2, cf2_0, cf3_2, C^2)$
$CF(t_4) = (cf1_2, cf2_0, cf3_2, C^2) \models CF(t_5) = (cf1_2, cf2_1, cf3_3, C^3)$
$CF(t_5) = (cf1_2, cf2_1, cf3_3, C^3) \models CF(t_6) = (cf1_3, cf2_1, cf3_3, C^4)$

It is easy to see that since the communication matrix has been modified in $CF(t_0) \models CF(t_1)$, according to definition 8 $CF(t_1)$ is reachable by one communicating step from $CF(t_0)$ or as it was indicated above $CF(t_0) \models^C CF(t_1)$. As the diagram suggests (the arrow going down from component P_3 to the matrix) a message has been

sent, and as a result $CF(t_0) \models^C CF(t_1) = CF(t_0) \models^S CF(t_1)$. Similarly $CF(t_2)$ is reachable by one processing step from $CF(t_1)$ that is $CF(t_1) \models^P CF(t_2)$ because the matrix is the same in both configurations.

Moreover $CF(t_0) \models CF(t_2)$ is a change of configuration, which is composed of two single changes of configurations, namely $CF(t_0) \models^C CF(t_1)$ and $CF(t_1) \models^P CF(t_2)$. To prove this, it is easy to see that

$CF(t_0) \neq CF(t_2)$ and $\exists\ c_1 = C^0, c_2 = C^0$ and $c_3 = C^1$ such that
$\exists\ F1_0 \in \Phi_1 \mid q_1^0 \in F_1(q_1, F1_0)$ with $F1_0\ (cf1_0 = (x_1, m_1, y_1, c_1, h_1)) = cf1_1 = (t_1, x_1', m_1', y_1', c_2)$ and
$\exists\ F3_0 \in \Phi_3 \mid q_3^0 \in F_3(q_3, F3_0)$ with $F3_0\ (cf3_0 = (x_3, m_3, y_3, c_2, h_3)) = (cf3_1 = (t_3, x_3', m_3', y_3', c_3))$ and
$cf2_0 = cf2_0$ with $c_2 = c_1$.

■

At this moment, a very interesting observation arises; we may easily check that the order in which $F1_0$ and $F3_0$ are executed is unimportant from the perspective of the whole change of configuration $CF(t_0) \models CF(t_2)$, thus the events that produce the changes are concurrent. Nevertheless, from definition 11, $CF(t_0) \models^* CF(t_3)$ is not a change of configuration, then there must be a relation of "dependency" between the events that trigger the changes from $CF(t_0)$ to $CF(t_3)$. A complete study of this relation of dependence/independence of events is presented in [61].

The key point about a change of configuration with respect to time according to [34] can be represented as follows. If t is the time when CF is reached and if $t' > t$ is the closest following instant at which a component finishes the execution of a function then CF' is the configuration at time t'. For the rest of the components that do not change their states this does not necessarily mean that such components have not done anything, but rather that they have not entirely completed executing a function. The incomplete executions of functions do not affect the complete ones mainly because the modifications of the communication matrix are done under mutual exclusion and the memories are local to each component. This fits with our assumption that every change of state and every modification to the communication matrix occur atomically at the end of each function. Changing the configuration $CF \models CF'$ implies that at least one machine has changed its state, a fact captured by our next;

Lemma 1: A *single change of configuration* is a *change of configuration*.

Proof. Let $CF \models^1 CF'$ be a single change of configuration. By definition 10, there must be one and only one component of the CSXMS that has changed its state and the communicating matrix may or may not be the same from the previous configuration. If the matrix has being changed then such a change only happens in one cell (see definition 8). Let p be the component that changes its state and let C and C' be the communicating matrices in each configuration.

Now if p is in state $(m, q, x, y, h::s, g)$ in CF and if $\phi \in \Phi$ is in p where $q' = F(q, \phi)$ then there are three cases:

- If $CF \models^p CF'$ then $\phi(x, m, y, C, h) = (t, x', m', y', C)$ results in the new state $cf'_P = (m', q', x', y', s, g::t)$ and since $\forall\ 1 \leq i \leq n, i \neq p\ cf_i = cf'_i$ and $C = C'$, by definition 11 $CF \models CF'$ is a change of configuration.
- If $CF \models^S CF'$ then $\phi(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{i,j} \Leftarrow y)$ and thus $cf'_P = (m, q', x, \lambda, s, g)$. Because $\forall\ 1 \leq i \leq n, i \neq p\ cf_i = cf'_i$ and since $C'[i, j] = y$ this implies that $\forall\ 1 \leq i < p, c_i = c_{i-1}, \forall\ p \leq i \leq n, c_i = c_{i-1}$ with $c_0 = C$ and $c_n = C'$ therefore by definition 11 $CF \models CF'$ is a change of configuration.
- If $CF \models^R CF'$ then $\phi(x, m, y, C, \varepsilon) = (\varepsilon, C[j, i], m, y, \Leftarrow C_{j,i})$ and thus $cf'_p = (m, q', x' = C[j, i], y, s, g)$. Because $\forall\ 1 \leq i \leq n, i \neq p\ cf_i = cf'_i$ and since $C'[j, i] = \lambda$ implying that $\forall\ 1 \leq i < p, c_i = c_{i-1}, \forall\ p \leq i \leq n, c_i = c_{i-1}$ with $c_0 = C$ and $c_n = C'$ therefore by definition 11 $CF \models CF'$ is a change of configuration.

□

However, as has already been mentioned, the opposite is not necessarily true (i.e. that a change of configuration is a single change of configuration) insofar as it is possible that several components change state in a single step (i.e. in the same change of configuration) like $CF(t_4) \models CF(t_5)$ in example 1.

Until now the formal model of CSXMS, allows us to speak basically of configurations and changes of them, nonetheless, in order to derive the output corresponding to an input sequence of symbols the concept of *relation computed* introduced in [31] is presented.

Definition 12: The *relation computed* by a CSXMS, $\boldsymbol{f}$: $(\Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_n^*) \leftrightarrow (\Gamma_1^* \times \Gamma_2^* \times \ldots \times \Gamma_n^*)$ is defined by:

$(s_1, s_2, \ldots, s_n)\, \boldsymbol{f}\, (g_1, g_2, \ldots, g_n) \Leftrightarrow \exists\ CF^0 = (cf_1, cf_2, \ldots, cf_n, C^0)$ and $CF' = (cf'_1, cf'_2, \ldots, cf'_n, C')$ with $\forall\ 1 \leq i \leq n$, $cf_i = (m_i, q_i, \lambda, \lambda, s_i, <>)$ and $cf'_i = (m'_i, q'_i, x'_i, y'_i, <>, g'_i)$ such that $CF^0 \models^* CF'$.

4.4 From CSXMS to EXM

The most remarkable achievement until now in the context of CSXMS from the testing point of view, establishes that it is possible to construct a stand-alone EXM from a CSXMS with the same input-output relationship. This result in principle permits the application and extension of the existing testing SXM methods to CSXMS. The construction of that result is due to Balanescu *et al.* [31]. Following their theorem 1, it can be presented here as:

Lemma 2: For every CSXMS there is an EXM, which computes the same input-output relation.

Proof. The construction of an EXM from a CSXMS $W_n = (R, CM, C^0)$ can be done as follows:

$X = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ where;

- $M = \prod_{k=1}^{n} (M_k \times IN_k \times OUT_k) \times CM$
- $\Sigma = \prod_{k=1}^{n} (\Sigma_k \cup \{\varepsilon\})$ and $\Gamma = \prod_{k=1}^{n} (\Gamma_k \cup \{\varepsilon\})$
- $Q = \prod_{k=1}^{n} Q_k$
- $\Phi = \prod_{k=1}^{n} (\Phi_k \cup \{\varepsilon\})$, where $\Phi : M \times \Sigma \leftrightarrow \Gamma \times M$, the type Φ is of the form;

  $(\phi_1, \phi_2, \ldots, \phi_n)[((m_1, in_1, out_1), \ldots, (m_n, in_n, out_n), C), (h_1, \ldots, h_n)] = ((t_1, \ldots, t_n), ((m'_1, in'_1, out'_1), \ldots, (m'_n, in'_n, out'_n), C')$

  1. if for any Λ_k either $\phi_k = h_k = t_k = \varepsilon$ (i.e. no function is applied) or $\phi_k(x_k, m_k, y_k, C, h_k::s_k^*) = (g_k^*::t_k, x_k', m_k', y_k', C')$ (i.e. a function was defined and applied) and
  2. if for all Λ_i, Λ_j where $\phi_i \neq \varepsilon$, $\phi_j \neq \varepsilon$ then ϕ_i and ϕ_j are not communicating functions corresponding to the same matrix location.

- $F : Q \times \Phi \rightarrow P(Q)$ is defined as $(q'_1, q'_2, \ldots, q'_n) \in F((q_1, q_2, \ldots, q_n), (\phi_1, \phi_2, \ldots, \phi_n))$
  If $\phi_k = \varepsilon$ then $q'_k = q_k$ otherwise $q_k \in F_k(q_k, \phi_k)$
- $I = \{(q_1, q_2, \ldots, q_n) \mid q_k \in I_k\}$ and $T = \{(q_1, q_2, \ldots, q_n) \mid q_k \in T_k\}$

From this construction the following result for the EXM and the CSXMS can be obtained directly:

$CF = (cf_1, cf_2, \ldots, cf_n, C) \models CF' = (cf'_1, cf'_2, \ldots, cf'_n, C')$ if and only if $u \vdash v$, where CF, CF' are configurations of the CSXMS and u and v are states of the EXM where a change of state $u = ((m_1, in_1, out_1), \ldots, (m_n, in_n, out_n), C), (q_1, \ldots, q_n), (s_1, \ldots, s_n), (g_1, \ldots, g_n)) \vdash v = ((m'_1, in'_1, out'_1), \ldots, (m'_n, in'_n, ou't_n), C'), (q'_1, \ldots, q'_n), (s'_1, \ldots, s'_n), (g'_1, \ldots, g'_n))$ is possible if

- $s_k = h_k s'_k$, $h_k \in \Sigma_k \cup \{\varepsilon\}$ and
- $\exists\ \phi = (\phi_1, \phi_2, \ldots, \phi_n) \in \Phi$ emerging from $(q_1, \ldots, q_n)$ and reaching $(q'_1, \ldots, q'_n)$ in such a way that $\phi[((m_1, in_1, out_1), \ldots, (m_n, in_n, out_n), C), (h_1, \ldots, h_n)] = ((t_1, \ldots, t_n), ((m'_1, in'_1, out'_1), \ldots, (m'_n, in'_n, out'_n), C')$ and $g'_k = g_k t_k$ with $t_k \in \Gamma_k \cup \{\varepsilon\}$.

□

It is not hard to see that the EXM obtained from the previous algorithm works in the same way as the CSXMS, where all component functions ϕ_k act in parallel.

It is important to remark that in the statements of lemma 2 the term "SXM" that appears in the original demonstrations has been replaced by the more accurate "EXM". The reason for doing this is that the resulting machine may support empty-operations [33, 60], but before proving this claim, we need the proper definition of an empty-operation.

Definition 13: An *empty-operation* $\phi \in \Phi_e$ of an EXM with type of the form Γ x M x Σ is defined by,

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

Or in a shorter form $\forall\, g^* \in \Gamma^*, \forall\, m \in M, \forall\, s^* \in \Sigma^*, \phi(m, \varepsilon) = (\varepsilon, m')$, where the symbol ε indicates that ϕ changes neither the input stream nor the output stream, and Φ_e is called the set of empty operations. We are now able to formulate the following result.

Lemma 3: The EXM obtained from a CSXMS following the algorithm of lemma 2 may perform empty-operations.

Proof. Let $CF = (cf_1, cf_2, \ldots, cf_n, C) \models CF' = (cf'_1, cf'_2, \ldots, cf'_n, C')$ be a change of configuration of the CSXMS such that for all $cf_i \neq cf'_i$ either (as in definition 8):

- $q' \in F_i(q, mvo_{ij})$ with $y \neq \lambda$ and $C[i, j] = \lambda$, where $mvo_{ij}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{i,j} \Leftarrow y) \in MVO_i$, thus $cf'_i = (m, q', x, y' \leftarrow \lambda, s, g)$ and $C'[i, j] = y$, or
- $q' \in F_i(q, mvi_{ji})$ with $x = \lambda$ and $C[j, i] \neq \lambda$, where $mvi_{ji}(x, m, y, C, \varepsilon) = (\varepsilon, C[j, i], m, y, \Leftarrow C_{j,i}) \in MVO_i$, thus $cf'_i = (m, q', x' \leftarrow C[j, i], y, s, g)$ and $C'[j, i] = \lambda$.

Now let $u \vdash v$ be the change of state in the EXM that corresponds to the change of configuration $CF \models CF'$ of the CSXMS, where the EXM was obtained by the procedure of lemma 2 from the CSXMS. Then because only communication functions have been applied, all the input and output streams of the components have been left unchanged, thus no change will be produced in the input or output of the EXM when the change $u \vdash v$ takes place, and so it will have performed an empty-operation.

□

Nevertheless, Ipate et *al.* [33] have suggested a simple abstract transformation for the communicating machines of a CSXMS so the entire system behaves as a SXM. This modification is called the CSXMS *testing-variant* ($CSXMS_n^T$) and is presented in section 5, but before this let us give a concrete example of how empty-operations might occur in the EXM resulting from the transformation described above.

Example 2: Figure 10 represents part of the machine obtained from the CSXMS for the farmer process of section 4.2 with one worker.

The states of the EXM are triples of the form $<q_i, q_j, q_k>$ where q_i corresponds to the state of the farmer, q_j presents the state of the worker and q_k is the state of the reaper. Each of the functions of the machine is denoted as $\phi = [\phi_i, \phi_j, \phi_k] \in \Phi$ where ϕ_i, ϕ_j and ϕ_k represent the functions that are applied in parallel from the farmer, the worker and the reaper and δ denotes that a component does not change its state.

In this way starting from state $<q_0, q_0, q_0>$ it is possible to reach state $<q_1, q_1, q_0>$ by means of the application of ϕ_1 = [*reads_pair*, *reads*, δ]. What this means is that if all the processes are in their initial state and if the functions *reads_pair* and *reads* are applied at the end the farmer and the worker will be in their corresponding states q_1. Indeed, this transition of the machines corresponds to a change of configuration of the overall system. It is not hard to see that in ϕ_1 the symbol $h_1 = (h_i, h_j, \varepsilon)$ is consumed from the input stream where clearly h_i and h_j are the first symbols in the input streams of the farmer and the worker. In addition, ϕ_1 will concatenate at the end of the output stream the symbol $t_1 = (t_i, t_j, \varepsilon)$ which is constructed in the same way as described above.

It might seem strange that in function $\phi_2 = [mvo_{F->K}, mvi_{K->F}, \delta]$ we have in the same change of configuration a pair of complementary communication operations, that is a send and its corresponding receive. However, there is nothing wrong with this since definition 11 allows us to do that. To prove this claim, let $<q_1, q_1, q_0>$ be the state of the EXM and let $CF = (cf_R, cf_F, cf_K, C)$ be the corresponding configuration of the overall system where cf_R, cf_F and cf_K are the states of the components reaper, farmer and worker in that order with $cf_F = (m, q_1, x, y, s, g)$, $cf_K = (m, q_1, x, y, s, g)$ for the farmer and the worker and $cf_R = (m, q_0, x, y, s, g)$ for the reaper.

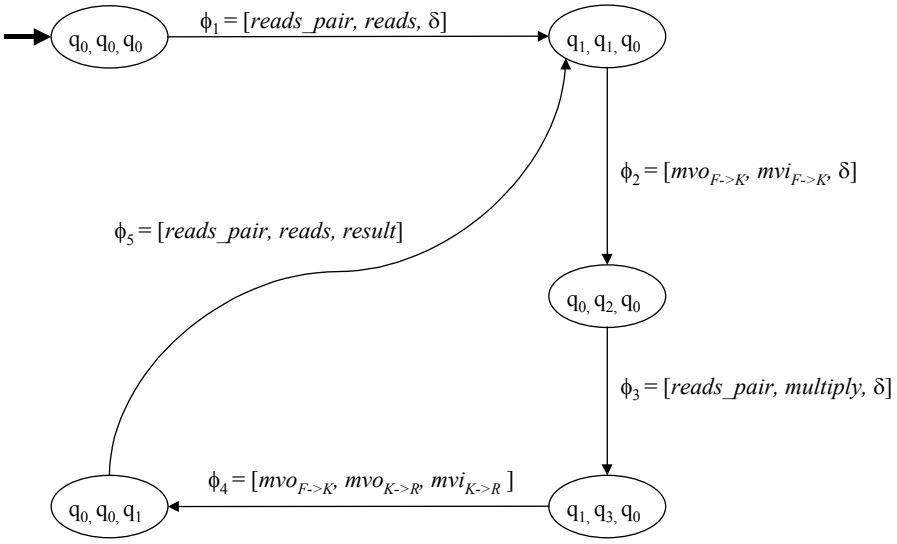


Figure 10

If we apply $mvo_{F->K}$ we obtain a new configuration $CF'' = (cf_R, cf'_F, cf_K, C'')$ where $cf'_F = (m, q_0, x, \lambda, s, g)$, $q_0 \in F_F(q_1, mvo_{F->K})$, $y \neq \lambda$, $C[F, K] = \lambda$, $mvo_{F->K}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{F,K} \Leftarrow y)$ and $C'' = (C_{F,K} \Leftarrow y)$. Now, from $CF'' = (cf_R, cf'_F, cf_K, C'')$ it is possible to apply $mvi_{F->K}$ since $q_2 \in F_K(q_1, mvi_{F->K})$, $x = \lambda$, $C''[F, K] \neq \lambda$ and $mvi_{F->K}(x, m, y, C'', \varepsilon) = (\varepsilon, C''[F, K], m, y, \Leftarrow C''_{F,K})$ resulting in configuration $CF'' = (cf_R, cf'_F, cf'_K, C')$ where $C'[F, K] = \lambda$ and and $cf'_K = (m, q_2, x, \lambda, s, g)$.

Finally, $\exists\ c_0, c_1, c_2 \in CM$ such that $c_0 = C$, $c_1 = C''$ and $c_2 = C'$ according to definition 11, $CF \models CF'$ is a change of configuration. Then by lemma 2, the change of state in the EXM $<q_1, q_1, q_0> \vdash <q_0, q_2, q_0>$ is possible by the application of $\phi_2 = [mvo_{F->K}, mvi_{K->F}, \delta]$. It is easy to see that ϕ_2 neither removes a symbol from the input stream nor concatenates a symbol to the output stream and thus ϕ_2 is an empty-operation and clearly, the EXM of figure 10 is not a SXM.

■

5. The testing variant of a communicating (Eilenberg) stream X-machines system ($CSXMS_n^T$)

The $CSXMS_n^T$ is obtained from a CSXMS by the introduction of new special input and output symbols, the most important aspect being the association of them with the communicating functions. Thus, any event performed by a component will consume and produce exactly one symbol, in this way ensuring that the whole system behaves like a SXM. This construction and all the results and definitions of the current section are taken from [33].

5.1 Obtaining a $CSXMS_n^T$

Definition 14: A system $W_n^T = (R^T, CM, C^0)$ is called a $CSXMS_n^T$ of a CSXMS $W_n = (R, CM, C^0)$, if R^T is obtained from R as follows:

- For all $1 \leq i \leq n$, $P_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0) \in R$ we have $P_i^T = (\Lambda^T_i, IN_i, OUT_i, in_i^0, out_i^0) \in R^T$ where $\Lambda_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i0})$ and $\Lambda_i^T = (\Sigma_i^T, \Gamma_i^T, Q_i, M_i, \Phi_i^T, F^T_i, I_i, T_i, m_{i0})$ with
- $\Sigma_i^T = \Sigma_i \cup \{\alpha\}$ where $\alpha \notin \bigcup_{j=1}^{n} \Sigma_j$.
- $\Gamma_i^T = \Gamma_i \cup \{1, 2, \ldots, k_i\}$ where $k_i = |\Phi''_i|$ and $o_i : \Phi''_i \rightarrow \{1, 2, \ldots, k_i\}$ is a one-to-one mapping.
- $\forall\ \phi_i''(x, m, y, C) = (x', m, y', C') \in \Phi''_i$ rewritten as $\phi_i''(x, m, y, C, \varepsilon) = (\varepsilon, x', m, y', C')$ and replaced by its testing variant $\phi_i''^T(x, m, y, C, \alpha) = (o_i(\phi_i''), x', m, y', C')$ in Φ''^T_i and $\Phi_i^T = \Phi'_i \cup \Phi''^T_i$.

To continue with the same notation P_i^T and Λ_i^T are respectively called the *testing-variant* of P_i and Λ_i. Additionally, a new set of matrices associated with *CM* is introduced and the concept of change of configuration is adapted by the following two definitions:

Definition 15: Let $\chi : CM \rightarrow CM^l$ be a one-to one mapping that produces the associated matrix C^l of a given *C*. CM^l is the set of matrices of order *n* x *n* associated with *CM* in the following way:

$\chi(C) \rightarrow C^l$ where $C \in CM$ and $C^l \in CM^l$ such that
if $i = j$ or $C[i, j] = \lambda$ then $C^l[i, j] = C[i, j]$
otherwise $C^l[i, j] = 1$

The associated matrix C^l of *C* is employed to distinguish states in the SXM constructed from a $CSXMS_n^T$. Let P_i be in state q_i for all $1 \leq i \leq n$ and let *C* be the communicating matrix then the SXM will be in state $((q_1, q_2,\ldots, q_n), C^l)$.

Definition 16: A *change of configuration* of a $CSXMS_n^T$ is defined as

$CF = (cf_1, cf_2,\ldots, cf_n, C) \models CF' = (cf'_1, cf'_2,\ldots, cf'_n, C')$ where $\forall\ 1 \leq i \leq n$, $cf_i = (m_i, q_i, x_i, y_i, s_i, g_i)$ and $cf'_i = (m'_i, q'_i, x'_i, y'_i, s'_i, g'_i)$ It is possible if $CF \neq CF'$ and $\exists\ c_0, c_1, \ldots, c_n \in CM$ with $c_0 = C$ and $c_n = C'$ such that $\forall\ 1 \leq i \leq n$ either:

- $cf_i = cf'_i$ and $c_i = c_{i-1}$ or
- $\exists\ \phi_i \in \Phi_i^T$ such that $q_i' \in F_i^T(q_i, \phi_i)$ and $\phi_i(x_i, m_i, y_i, c_{i-1}, h_i) = (t_i, x'_i, m'_i, y'_i, c_i)$ where $s_i = h_i::s'_i$, $g'_i = g_i::t_i$, $h_i \in \Sigma_i^T$ and $t_i \in \Gamma_i^T$.

With these modifications, it is now possible to demonstrate that the $CSXMS_n^T$ behaves as a SXM. The next result is taken from theorem 5.1 in [33].

Lemma 4: For every CSXMS there is a SXM X^T, which computes the same input-output relationship as the $CSXMS_n^T$ of the CSXMS.

Proof. The construction of X^T from the $CSXMS_n^T = (R^T, CM, C^0)$ can be done as follows:

$X^T = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ where:

- $M = \prod_{k=1}^{n} (IN_k \times M_k \times OUT_k) \times CM$. The initial memory value is $m_0 = ((\lambda, m_{10}, \lambda),\ldots, (\lambda, m_{n0}, \lambda), C^0)$
- $\Sigma = \prod_{k=1}^{n} (\Sigma_k^T \cup \{\varepsilon\}) - (\varepsilon,\ldots, \varepsilon)$
- $\Gamma = \prod_{k=1}^{n} (\Gamma_k^T \cup \{\varepsilon\}) - (\varepsilon,\ldots, \varepsilon)$
- $Q = \prod_{k=1}^{n} Q_k \times \chi(CM)$
- $\Phi = \prod_{k=1}^{n} (\Phi_k \cup \{\delta\}) \mid (\phi_1, \phi_2,\ldots, \phi_n) \neq (\delta, \delta,\ldots, \delta)$ and $\neg\exists\ i, j$ such that $\phi_i = mvo_{i->j}$ and $\phi_j = mvi_{i->j}$

  where $\Phi : M \times \Sigma \leftrightarrow \Gamma \times M$ stands for all $\phi_i \in \Phi_i^T \cup \{\delta\}$ acting in parallel and $\phi_i = \delta$ denotes that P_i does not change its state. The type Φ is of the form:

  $(\phi_1, \phi_2,\ldots, \phi_n)[((in_1, m_1, out_1),\ldots, (in_n, m_n, out_n), C), (h_1,\ldots, h_n)] = ((t_1,\ldots, t_n), ((in'_1, m'_1, out'_1),\ldots, (in'_n, m'_n, out'_n), C')$
  if for any Λ_k any
  $\phi_k = \delta$ (i.e. no function is applied), or
  $\phi_k(x_k, m_k, y_k, C, h_k::s_k^*) = (g_k^*::t_k, x_k', m_k', y_k', C')$ (i.e. a function was applied) and

if for all Λ_i, Λ_j where $\phi_i \neq \delta$, $\phi_j \neq \delta$ then ϕ_i and ϕ_j are not communicating functions corresponding to the same matrix location.

- F : Q x $\Phi \rightarrow$ P(Q) is defined as $(q'_1, q'_2, \ldots, q'_n, C^l) \in (F((q_1, q_2, \ldots, q_n), C'^l), (\phi_1, \phi_2, \ldots, \phi_n))$
  If for all ϕ_k whichever:
  $\phi_k = \delta$ and $q'_k = q_k$ or
  $\phi_k \in \Phi_i'$ and $q'_k \in F_k(q_k, \phi_k)$ or
  $q'_k \in F_k(q_k, \phi_k)$ and $\phi_k = mvo^T_{K->j}$, $C^l[k, j] = \lambda$ and $C'^l[k, j] = 1$ for some $j \neq k$ or
  $q'_k \in F_k(q_k, \phi_k)$ and $\phi_k = mvi^T_{j->K}$, $C^l[j, k] = 1$ and $C'^l[k, j] = \lambda$ for some $j \neq k$
- I = $\{((q_1, q_2, \ldots, q_n), C^0) \mid \forall\ 1 \leq k \leq n, q_k \in I_k\}$ and T = $\{((q_1, q_2, \ldots, q_n), C^l) \mid \forall\ 1 \leq k \leq n, q_k \in T_k\}$

X^T is called the equivalent SXM of the $CSXMS_n^T$. From the above construction, it follows that;

$CF = (cf_1, cf_2, \ldots, cf_n, C) \models CF' = (cf'_1, cf'_2, \ldots, cf'_n, C') \Leftrightarrow$ if $u \vdash v$, where
CF, CF' are configurations of the $CSXMS_n^T$ and u and v are states in X^T.

□

Example 3: Let us consider the diagram in figure 11 where part of the SXM resulting from the $CSXMS_n^T$ for the farmer process is shown.

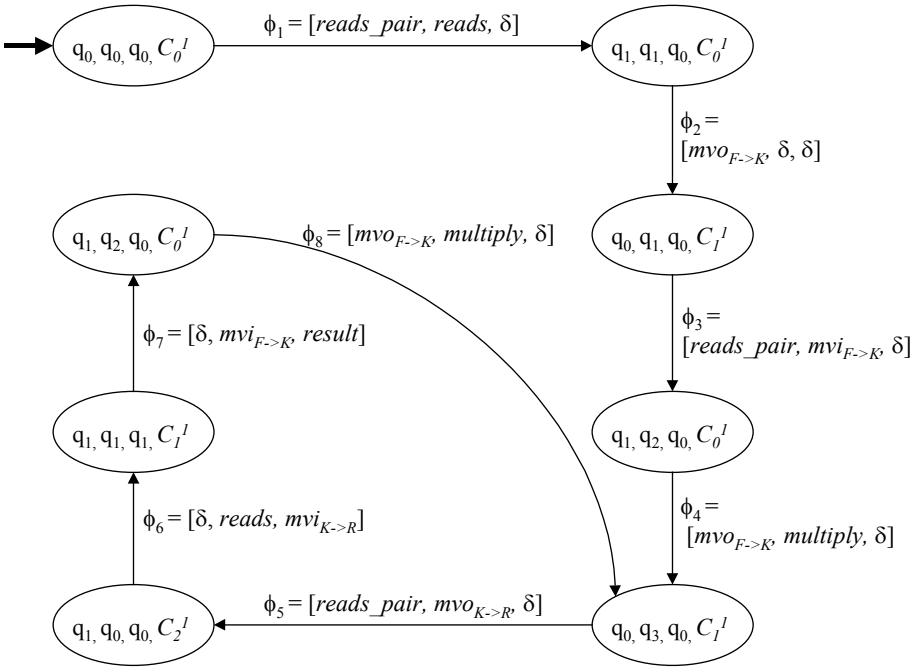


Figure 11

From lemma 4, the states of the SXM are of the form $<q_i, q_j, q_k, C^l>$ and the functions are of the form $\phi = [\phi_i, \phi_j, \phi_k] \in \Phi$, where q_i, q_j, q_k and ϕ_i, ϕ_j, ϕ_k are the states and the functions that are applied in parallel that correspond to the farmer, the worker and the reaper in that order. The matrix C^l is as in definition 15 and δ indicates that a component does not change its state (i.e. an empty sequence of functions).

In the same way as in section 4.4, it is possible for instance to reach state $<q_1, q_1, q_0, C_0^l>$ from $<q_0, q_0, q_o, C_0^l>$ by means of $\phi_1 = [reads\_pair, reads, \delta]$. Nevertheless, function $\phi_2 = [mvo_{F->K}, \delta, \delta]$ is now different from the function ϕ_2 of the previous example (section 4.4) since lemma 4 explicitly establishes that for any $\phi = (\phi_1, \phi_2, \ldots, \phi_n)$ there is no pair i, j to enable $\phi_i = mvo_{i->j}$ and $\phi_j = mvi_{i->j}$. In other words, it is not possible (as before) to have in the same change of configuration two complementary communicating send/receive functions. Moreover, here $\phi_2 = [mvo_{F->K}, \delta, \delta]$ will consume and produce one symbol even in the case when just one communicating function is performed, because the original $mvo_{F->K}$ is replaced by its testing variant $mvo_{F->K}(x, m, y, C, \alpha) = (o_i(\phi_i''), x, m, \lambda, (C_{FK} \Leftarrow y))$.

■

5.2 Properties of the $CSXMS_n^T$ components with respect to the equivalent SXM

In this section some interesting properties of the components of a $CSXMS_n^T$ that are reflected in the equivalent SXM are proved.

First of all, in the context of SXM testing, the concept of determinism has played a fundamental role since the core of its method has been supported up to now under the hypothesis of deterministic behaviour for the machine models [10, 43]. However, recent papers have addressed the issue of non-determinism for both the specification [55, 58] and the implementation [54, 57, 59]. Although in [56] it is demonstrated that determinism is not really a restriction for the application of this approach, it is important to consider it here, because determinism may reduce computational effort. In other words, for any non-deterministic SXM there is another SXM for which its associated FA is deterministic (DFA), and which computes the same relationship. This implies that if the associated FA of one machine is non-deterministic (NFA), some computation is needed in order to derive an equivalent deterministic machine; hence, an extra workload might be required.

The basic idea of determinism with respect to the associated FA of a given SXM can be found in definition 24 of [27] among other references [10, 43].

Lemma 5: If for all components P_i of the system, the associated FA of Λ_i, is DFA then the associated FA of X^T is also a DFA.

Proof. Let $A^T = (\Phi, Q, F, I, T)$ be the associated automaton of X^T.

Now, if for all $P_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0) \in R^T$ with $\Lambda_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i0})$ we have that $I_i = \{q_{i0}\}$, it follows that $|(\{((q_1, q_2,\ldots, q_n), C^0)$ such that $q_i \in I_i\})| = 1$ (i.e. the cardinality is one). In other words, there is only one initial state for X^T and thus for A^T.

For the second requirement let us assume the contrary, that is $\exists\, q, q', q'' \in Q$ and $\exists\, \phi \in \Phi$ for A^T such that $\{q', q''\} \subseteq F(q, \phi)$. If $q = ((q_1, q_2,\ldots, q_n), C^l)$, $q' = ((q'_1, q'_2,\ldots, q'_n), C'^l)$ and $q'' = ((q''_1, q''_2,\ldots, q''_n), C''^l)$ and if $\phi = (\phi_1, \phi_2,\ldots, \phi_n)$ it is direct that if $q' \neq q''$ then there is at least one $1 \leq i \leq n$ with $q'_i \neq q''_i$ and $\{q'_i, q''_i\} \subseteq F_i(q_i, \phi_i)$ which gives a contradiction if the associated automaton Λ_i, of P_i is deterministic.

□

Lemma 6: If, for all components P_i of the system, Φ'_i is a set of (partial) functions then for X^T we have that Φ is also a set of (partial) functions.

Proof. For all $P_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0) \in R^T$ with $\Lambda_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i0})$ because $\Phi_i = \Phi'_i \cup \Phi''_i$, from definition 14, Φ''^T_i is clearly a set of (partial) functions, so is Φ'_i due to the fact that $\Phi_i^T = \Phi'^T_i \cup \Phi''^T_i$ thus Φ_i^T must be a set of (partial) functions. Finally, from lemma 4 it is easy to see that Φ of X^T has to be a set of (partial) functions.

□

For the following lemmas, let us say:

$M_i' \subseteq M_i$,
$M_i'' = IN_i \times M_i' \times OUT_i$,
$M' = M_1'' \times M_2'' \times \ldots \times M_n'' \times C$ and
$S_i = \{S_i^{\phi i} \mid \phi_i \in \Phi_i'\}$ where
$S^{\phi i} \subseteq \Sigma_i$, $S = \{S^\phi \mid \phi \in \Phi\}$,
$S^\phi = (S_1^{\phi 1}, S_2^{\phi 2}, \ldots, S_n^{\phi n})$ and
If $\phi_i \in \Phi_i''^T$ then $S_i^{\phi i} = \{\alpha\}$ and $S_i^\delta = \{\delta\}$.

For a better understanding of the particular notation using in the following lemmas, the reader should refer to [27].

Lemma 7: If, for all components P_i of the system, Φ_i' is closed with respect to (M_i', S_i) then for X^T we have that Φ is closed with respect to (M', S) (see definition 30 in [27]).

Proof. If in X^T we have that $\phi(m, h) = (t, m')$ where:

$\phi = (\phi_1, \phi_2, \ldots, \phi_n) \in \Phi$,
$m = ((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n), C) \in M'$,
$m' = ((in'_1, m'_1, out'_1), \ldots, (in'_n, m'_n, out'_n), C') \in M$,
$h = (h_1, \ldots, h_n) \in S^{\phi}$,
$t = (t_1, \ldots, t_n) \in \Gamma$

then for all $1 \leq i \leq n$, either:

If $\phi_i \in \Phi_i'$ (is a processing function) since $\Phi_i' \otimes (M_i', S_i)$ then
$m'_i \in M_i'$ implying that $(in'_i, m'_i, out'_i) \in M_i''$ or
If $\phi_i \in \Phi''_i{}^{T}$ (is a communicating function) or $\phi_i = S_i^{\delta}$ (no function was applied) then
$m_i = m'_i$ therefore $(in'_i, m'_i, out'_i) \in M_i''$

Therefore $m' \in M'$ and clearly $\Phi \otimes (M', S)$

□

Lemma 8: If for all components P_i of the system Φ_i' is output-distinguishable with respect to (M_i', S_i) then for X^T we have that Φ is output-distinguishable with respect to (M', S) (see definitions 33, 37 in [27]).

Proof. If in X^T we have that $\phi(m, h) = (t, m')$ and $\varphi(m, h) = (t, m'')$ where:

$\phi = (\phi_1, \phi_2, \ldots, \phi_n)$, $\varphi = (\varphi_1, \varphi_2, \ldots, \varphi_n) \in \Phi$,
$m = ((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n), C) \in M'$,
$m' = ((in'_1, m'_1, out'_1), \ldots, (in'_n, m'_n, out'_n), C'), \in M'$,
$m'' = ((in''_1, m''_1, out''_1), \ldots, (in''_n, m''_n, out''_n), C'') \in M'$,
$h = (h_1, \ldots, h_n) \in S^{\phi} \cap S^{\varphi}$,
$t = (t_1, \ldots, t_n) \in \Gamma$

in that case $C'_0, C'_1, \ldots, C'_n, C''_0, C''_1, \ldots, C''_n \in CM$ exists with $C'_0 = C''_0 = C$, $C'_n = C'$ and $C''_n = C''$ such that for all $1 \leq i \leq n$,

$\phi_i(((in_i, m_i, out_i), C'_{i-1}), h_i) = (t_i, ((in'_i, m'_i, out'_i), C'_i))$ and
$\varphi_i(((in_i, m_i, out_i), C''_{i-1}), h_i) = (t'_i, ((in''_i, m''_i, out''_i), C''_i))$

where $t_i = t'_i$

Without losing the generality of the argument, let us suppose that $\phi_i \in \Phi_i'$ (a processing function) and $\varphi_i \in \Phi''_i{}^{T}$ (a communicating function). However, this implies that the output symbol t_i, produced by ϕ_i is different from the output symbol t'_i produced by φ_i, which is a contradiction. Again without loss of generality, let $\phi_i = S_i^{\delta}$ and $\varphi_i \neq S_i^{\delta}$ but this is also a contradiction $t_i \neq t'_i$. As a result, there are three cases:

If $\phi_i, \varphi_i \in \Phi_i'$ (are processing functions) clearly since Φ_i' is output-distinguishable then
$\phi_i = \varphi_i$, $in'_i = in''_i$, $m'_i = m''_i$, $out'_i = out''_i$ and $C'_i = C''_i$
If $\phi_i, \varphi_i \in \Phi''_i{}^{T}$ (are communicating functions) from the construction of $\Phi''_i{}^{T}$ then
given that $t_i = t'_i$ and since $t_i = o_i(\phi_i'')$ and $t'_i = o_i(\varphi_i'')$, from definition 14 necessarily $\phi_i'' = \varphi_i''$
as a consequence $\phi_i = \varphi_i$, $in'_i = in''_i$, $m'_i = m''_i$, $out'_i = out''_i$ and $C'_i = C''_i$
If $\phi_i, \varphi_i \in S_i^{\delta}$ (neither function was applied) then trivially
$\phi_i = \varphi_i$, $in'_i = in''_i$, $m'_i = m''_i$, $out'_i = out''_i$ and $C'_i = C''_i$

Hence, if $\phi(m, h) = (t, m')$ and $\varphi(m, h) = (t, m'')$ for X^T then $\phi = \varphi$ and $m' = m''$

□

5.3 The simple CSXMS (CSXMS-s)

The last two lemmas show that, if output-distinguishability is present in all the components of a $CSXMS_n^T$ then the same property is inherited at the system level by the SXM X^T constructed as in lemma 4. For the case of completeness, a class of CSXMS has been identified in [33] for which the property of input-completeness is

preserved in X^T when completeness is there in all the components of the system. This type of system is called a CSXMS-s and the features of its components are:

1. All the initial states of the components must be processing states,
2. All the processing functions empty the input port and compute a non-empty value for the output port.
3. All communicating functions change the machine to a processing state. Put differently, no two or more consecutive communications can be executed for a component, within some intervening processing functions also being executed.

These properties can be stated formally as follows.

Definition 17: A component $P_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0)$ with $\Lambda_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i0})$ of a CSXMS is simple if

- $I_i \subseteq Q'_i$.
- $\forall\ \phi' \in \Phi'_i : \phi'(in_i, m_i, out_i, C_i, h_i) = (t_i, in'_i, m'_i, out'_i, C_i)$ then $in_i' = \lambda$ and ($out'_i \neq \lambda$ or $OUT_i = \{\lambda\}$)
- $\forall\ \phi'' \in \Phi''_i, q_i \in Q''_i : F_i(q_i, \phi'') = q$ then $q \in Q'_i$

Definition 18: In a CSXMS-s all the components P_i of the system are simple.

Lemma 9: For any component P_i of a CSXMS-s W_n, if $\phi_i \in \Phi''_i$ where $\phi = (\phi_1, \phi_2, \ldots, \phi_n) \in \Phi$, $m = ((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n), C) \in Mattain_\phi(W_n)$ then $in_i = \lambda$ and ($out_i \neq \lambda$ or $OUT_i = \{\lambda\}$).

Proof. Since $m \in Mattain_\phi(W_n)$ from definition 28 in [27] there must be a $q = ((q_1, q_2, \ldots, q_n), C) \in Q$ such that $Mattain_q(W_n)$. From lemma 4 it follows that $q_i \in Q''_i$ and because W_n is simple $q_i \notin I_i$ and $\forall\ \phi \in \Phi_i$, $q \in Q_i$ such that $F_i(q, \phi) = q_i$ we have that $\phi \in \Phi'_i$. For the reason that $m \in Mattain_q(W_n)$ it is undoubted that $in_i = \lambda$ and ($out_i \neq \lambda$ or $OUT_i = \{\lambda\}$)

□

Lemma 10: If, for all components P_i of the system, Φ_i' is complete with respect to (M_i', S_i) then for X^T we have that Φ is input-complete with respect to (M', S) (see definition 31, 36 in [27]).

Proof. If in X^T we have $\phi(m, h)$ where:

$\phi = (\phi_1, \phi_2, \ldots, \phi_n) \in \Phi$,
$m = ((in_1, m_1, out_1), \ldots, (in_n, m_n, out_n), C) \in Mattain_\phi(W_n) \cap M'$

consequently, for all $1 \leq i \leq n$, either:

If $\phi_i \in \Phi_i'$ then
  $(in_i, m_i, out_i) \in M_i''$ implying that $m_i \in M_i'$ and clearly $\exists\ h_i \in S_i^{\phi i}$ such that
  $\phi_i((in_i, m_i, out_i, C), h_i) \neq \varnothing$ or
If $\phi_i \in \Phi''_i$ was transformed (definition 14) into $\phi_i^T \in \Phi''^T_i$ then any of the follows;
  $\phi_i = mvo_{i->j} \in MVO_i$ then $\exists\ j \mid j \neq i$ and $C[i, j] = \lambda$,
  as $m \in Mattain_\phi(W_n)$ then from lemma 9 $out_i \neq \lambda$ or
  $\phi_i = mvi_{j->i} \in MVI_i$ then $\exists\ j \mid j \neq i$ and $C[j, i] \neq \lambda$,
  since $m \in Mattain_\phi(W_n)$ then from lemma 9 $in_i = \lambda$.
  Clearly, $\phi_i((in_i, m_i, out_i, C), \varepsilon) \neq \varnothing$ and thus $\phi_i^T((in_i, m_i, out_i, C'), \alpha) \neq \varnothing$

Hence, $\exists\ h = (h_1, \ldots, h_n) \in S^\phi$ such that in X^T it is true that $\phi(m, h) \neq \varnothing$

□

6. The CSXMS with channels (CSXMS-c)

In the CSXMS-c model [34] the following design decisions are made with respect to the communication through the various channels:

1. Each channel has a single sender and a single receiver.

2. The communications involve a send/receive operation.

The operations (i.e. send/receive) use rendezvous-like synchronisation mechanisms similar to the ones found in Ada [62, 63]. Thus, when a communication operation is taking place, and when the first of the two processes (let us say the sender) is ready for a rendezvous (*i.e.* with the receiver). It will be blocked and so waits until the other (*i.e.* the receiver) reaches the point (*i.e.* transition) at which the reception of the message is specified (*i.e.* when the receiver is also ready for the rendezvous). Afterwards, the communication takes place and the two processes can continue to act. The same, in the opposite direction, will occur when the receiver is ready first.

The synchronisation mechanism introduced in the CSXMS-c does not need to make any assumption about the mutual exclusion of access to the elements of the communication matrix as in the CSXMS, mainly because the kind of communication presented here naturally assures such a condition. This section is based on the results obtained by [34].

6.1 The CSXMS-c model

Definition 19: The CSXMS-c model $W_n = (R', CM, C^0)$ for a CSXMS with n components $W_n = (R, CM, C^0)$ includes an additional communicating machine P_{n+1} such that $R' = R \cup P_{n+1}$. The new component is called the *communication server* or simply the *server* where:

- $IN_{n+1} \subseteq \{\lambda, @\} \cup \bigcup_{j=1}^{n} \{jS^+, jR^+, jS^-, jR^-\}$ and
- $OUT_{n+1} \subseteq \{\hookleftarrow\} \cup \bigcup_{j=1}^{n} \{jS^-, jR^-\}$

Besides this, for all the other components $P_i = (\Lambda_i, IN_i, OUT_i, in_i^0, out_i^0)\ \forall\ 1 \leq i \leq n$, the following applies:

- $IN_i \subseteq M_i \cup \{\lambda, \hookleftarrow\} \cup \bigcup_{j=1}^{n} \{jS^-, jR^-\}$ and
- $OUT_i \subseteq M_i \cup \{@, \hookleftarrow\} \cup \bigcup_{j=1}^{n} \{jS^+, jR^+\}$

The meaning of λ, as in the CSXMS model, is the absence of a message. The symbol @ indicates, again as in the CSXMS, a disconnected channel. However, the functionality of @ is extended in the CSXMS-c as follows; when a machine is going to stop, just before a final state is reached, the symbol @ is assigned to all of the cells of the row-column that correspond to that machine. The symbol jS^+ or jR^+ is sent from any machine to the server to request authorisation to send or receive a message to/from machine P_j. The server sends the symbol ↵ (called OK in [34]) to a machine if the required communication operation is authorised. By contrast, when an attempt to communicate is rejected by the server due to the fact that machine P_j is not yet ready to complete the operation, the symbols jS^- or jR^- are used.

Definition 20: A *server machine* is a 5-tuple $P_{n+1} = (\Lambda_{n+1}, IN_{n+1}, OUT_{n+1}, in_{n+1}^0, out_{n+1}^0)$ where the local memory M_{n+1} stores a subset $B \subseteq \{1, 2, \ldots, n\}$ such that $j \in B$ if and only if $C[j, n+1] \neq @$. The initial memory contains the whole set of values.

The operation of the server finishes when B is empty, before that it continues selecting an item each time from the memory. The choice can be done randomly or using a particular form of data organisation like lists, stacks, etc. Consequently, in the CSXMS-c definition that is presented here, it has been decided that the memory of the server is organised as $M = (\{\lambda, @, \hookleftarrow\} \cup \{jS^+, jR^+, j\ S^-, jR^-\})$ x B[N] x $\{1, 2, \ldots, n\}$ where the first component $a \in (\{\lambda, @, \hookleftarrow\} \cup \{jS^+, jR^+, j\ S^-, jR^-\}$ stores the last symbol received or the symbol that is going to be sent. The second component is an array B of n boolean values, one for each of the other components of the system. Initially, $\forall\ 1 \leq i \leq n$, $B[i] = true$, meaning that all the processes have not yet finished their corresponding tasks and once a process P_j has finished the value is changed in $B[j]$ to *false*. The third symbol $i \in \{1, 2, \ldots, n\}$ is a counter that controls the order in which the components of the CSXMS-c are taken (i.e. in this case in a round-robin mode) and clearly at the beginning $i = 1$. In what follows, if m represents the memory, $m.a$ denotes the first component, $m.B[m.i]$ means the *i-th* element in the array B of the memory and $m.i$ denotes the counter.

A major difference between the CSXMS and the CSXMS-c models is in the *type of channels* allowed between the server and the rest of the machines in the system. In effect, the matrix structure of the communication

subsystem for a CSXMS permits in principle the modelling of full-duplex channels, that is bi-directional and simultaneous communication can occur because a pair of machines P_i and P_j might access C[i, j] and C[j, i] at the same time. By contrast, in the CSXMS-c the communication between the server and any of the other machines is of the half-duplex type, in the sense that it is bi-directional but cannot occur at the same time in both directions. This property is achieved by means of d defined as the n + 1 column of the matrix and consequently d_i defined as C[i, n+1] $\forall$ 1 ≤ i ≤ n + 1. The communication between the machines and the server is accordingly controlled by the following design decisions:

1. During the execution of the protocols associated with the send/receive operations the machine P_i will use d_i with $i \neq n + 1$ as will be shown later.
2. When machine P_i stops, just before its final state is reached $\forall$ 1 ≤ j ≤ n + 1, C[i, j] = @ and C[j, i] = @. Obviously, after that d_i must have the @ symbol assigned to it.

This has an important bearing on the nature of the communicating functions of the CSXMS. Because $mvo_{i->j}$ and $mvi_{j->i}$ are in that order defined in function of the cells $C[i, j]$ and $C[i, j]$ of the matrix, it is impossible to achieve the previous design decisions of the CSXMS-c solely by means of combining them. For example, some of the CSXMS-c operations require access to cell d_i = C[i, n+1] from machine P_i not only in an output-move but also in an input-move. In [34] the operation of the server is formalised by means of the algorithm written in pseudo-code presented in table 1.

When the server considers the value i, it acts as follows:

```
case d_i of
        d_i = @          : delete i from B;
        d_i = jS+        : if d_i = iR+ then { d_i ← ↵; d_j ← ↵;}
                           else
                                   if d_j = iR- then –
                                   else d_i ← jS-
        d_i = jR+        : if d_j = iS+ then { d_i ← ↵; d_j ← ↵;}
                           else
                                   if d_i = iS- then –
                                   else d_i ← jR-
        else             : -
end
```

Table 1

However, in order to keep within the same framework as the rest of the document, we shall specify the server here using the same CSXMS conventions. The state-transition diagram for the server appears in figure 12.

The server operates on a loop using for that the memory counter $m.i$, which initially is 1.

Every time the value is incremented (in *next*, *is_@* and *to_q*$_0$), the % operator (i.e. modulo) is applied in an obvious manner. All the server functions are listed in table 2.

Following the above, when the server considers the value $m.i$ (after the function *choice* has been successfully performed) a message is read from machine P_i and it is stored in $m.a$ (in functions *read* and *input*). Then there are three cases:

1. If the symbol received is @, this means that machine P_i has already terminated its operation, therefore it is necessary to change m.B*[m.i]* to *false* (in *is_@* function). In this way, the next time the server control is in initial state q_0, machine P_i will no longer be considered for further communications in the system (in *next* or *stop* functions).
2. If the symbol received is jS$^+$ this means that machine *Pi* is requesting to send a message to machine *Pj*. Consequently, the communication function *read_j* is executed in order to check the status of *Pj* and again there are three cases:
    2.1 If the symbol received is iR$^+$, the machine P_j is asking to receive a message from P_i. That is, both sender and receiver are ready to perform a communication (i.e. a rendezvous takes place). Then the server sends the symbol ↵ to both machines through the functions *ok*, to_i and to_j to indicate that the server authorises the operation. After that, the machine continues its loop (in

to_q₀).

2.2 If the symbol is $i\mathrm{R}^-$ the server previously rejected an attempt of P_j to receive a message from P_i. If that is the case then the server simply continues with the next iteration of the loop, put differently the value *m.i* is incremented as it was described above and the machine moves to initial state q_0 (in *to_q₀* function).

2.3 If the symbol is neither $i\mathrm{R}^+$ nor $i\mathrm{R}^+$ then the server rejects the attempt of P_i to send a message to P_j. Therefore, the symbol $j\mathrm{S}^-$ is sent to P_i from the server and the machine moves to the next iteration of the loop. All this operation is performed using *other_r*, *no_send*, to_i and *to_q₀*.

3. If the symbol received is $j\mathrm{R}^+$ this means that machine *Pi* is requesting to receive a message from machine *Pj*. The communication function *read_j* is performed to evaluate the status of the sender and there are three cases, which are symmetrical to the complementary operations described for sending a message.

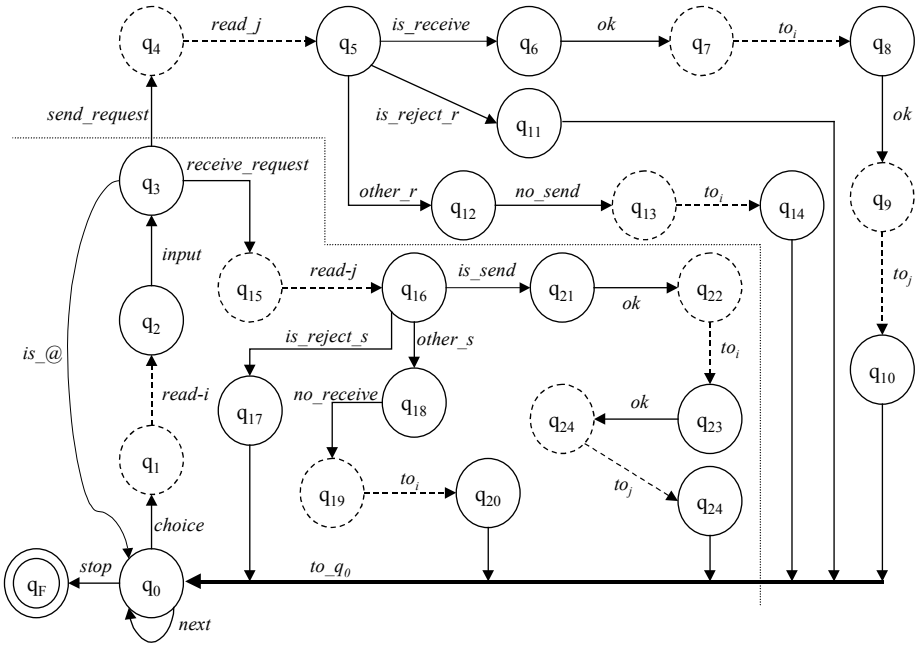


Figure 12

Already, then, we see that the nature of the communication functions for a CSXMS requires some extensions and in this direction, it must be noticed that both *read_i* and *read_j* operate in the same way as an ordinary input-move function. However, the communication functions to_i and to_j are different from an ordinary output-move function. To illustrate this, consider the moment when the server is sending a message to machine P_i, if such an operation would have been specified using an output-move as follows:

$$mvo_{n+1->i}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{n+1,m.i} \Leftarrow y), \text{ if } y \neq \lambda \text{ and } \mathrm{C}[n+1, m.i] = \lambda$$

then the cell C[n+1, i] would have been used to pass the message from the server to machine P_i. Instead, such an operation in the context of the CSXMS-c is defined as:

$$to_i(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{m.i,\, n+1} \Leftarrow y), \text{ if } y \neq \lambda \text{ and } \mathrm{C}[m.i, n+1] = \lambda$$

where the cell d_i = C[i, n+1] is utilised for establishing the two-way communication.

One aspect of the CSXMS-c notation has to do with making synchronisation explicit as a rendezvous-like mechanism between machines. Because of this, the operations for transmitting messages involve a number of interactions with the server and so they are described as macro-functions using the lower level functions of the CSXMS. On top of this, the **select** construct with guarded alternatives and the **terminate** clause were introduced. The alternatives of the form [**when** *cond* =>] *j* **!** *val* have the meaning: if *cond* is fulfilled, then *val* has to be sent to machine P_j. Figure 13.a schematises the next-state function for the sending form. The state names correspond to those presented in [34], and the new intermediate states that appear here are named by a letter *q* followed by

the index of the previous state to indicate that such a pair of states corresponds to the same state in [34]. The reason for these new intermediate states is because the specification has been done here using the lower level EXM notation.

$choice(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.B[m.i] = true$
$next(x, m, y, C, \varepsilon) = (\varepsilon, x, m'.i \leftarrow ((m.i + 1)\%(n)) + 1, y, C)$, if $m.B[m.i] = false$ and $\exists\, j \mid m.B[j] = true$
$stop(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $\forall\, j \mid m.B[j] = false$

$read\_i(\lambda, m, y, C, \varepsilon) = (\varepsilon, x' \leftarrow C[m.i, n+1], m, y, C)$, if $C[m.i, n+1] \neq \lambda$
$read\_j(\lambda, m, y, C, \varepsilon) = (\varepsilon, x' \leftarrow C[m.j, n+1], m, y, C)$, if $C[m.j, n+1] \neq \lambda$
$input(x, m, y, C, \varepsilon) = (\varepsilon, x, m'.a \leftarrow x, y, C)$, if $x \neq \lambda$
$is\_@(x, m, y, C, \varepsilon) = (\varepsilon, x, m'.B[m.i] \leftarrow false; m'.i = ((m.i + 1)\%(n)) + 1, y, C)$, if $m.a = @$
$send\_request(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.a = jS^+$
$receive\_request(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.a = jR^+$

$is\_receive(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.a = iR^+$
$is\_send(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.a = iS^+$
$is\_reject\_r(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.a = iR^-$
$is\_reject\_s(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.a = iS^-$
$other\ \ r(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.a \neq iR^+$ and $m.a \neq iR^-$
$other\_s(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if $m.a \neq iS^+$ and $m.a \neq iS^-$

$no\ \ send(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y' \leftarrow jS^-, C)$
$no\_receive(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y' \leftarrow jR^-, C)$
$ok(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y' \leftarrow \lrcorner, C)$

$to_i(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda,\ C_{m.j,\, n+1} \Leftarrow y)$, if $y \neq \lambda$ and $C[m.j, n+1] = \lambda$
$to_i(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda,\ C_{m.i,\, n+1} \Leftarrow y)$, if $y \neq \lambda$ and $C[m.i, n+1] = \lambda$
$to\_q_0(x, m, y, C, \varepsilon) = (\varepsilon, x, m'.i \leftarrow ((m.i + 1)\%(n)) + 1, y, C)$,

Table 2

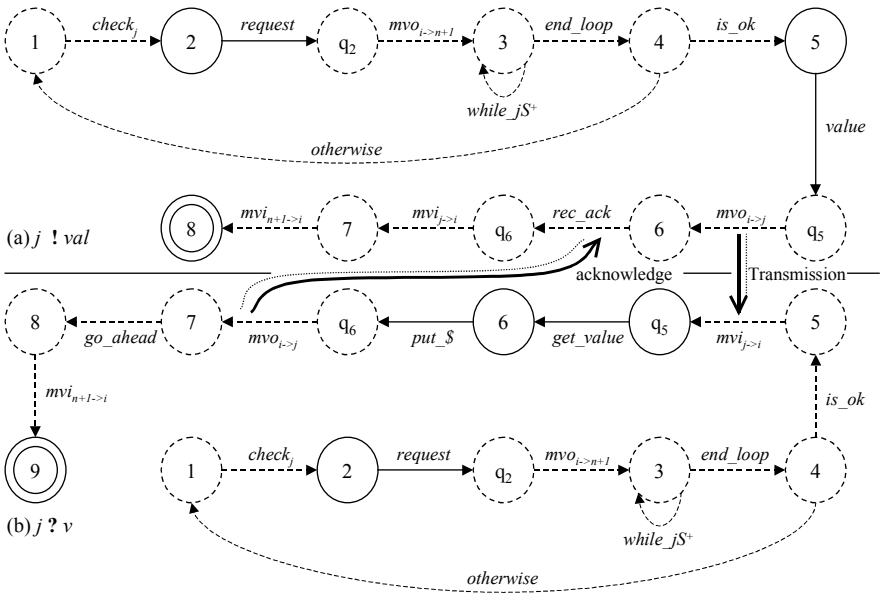


Figure 13

At the beginning, machine P_i evaluates if *cond* is true and if the contents of $C[i, j]$ are different from @. Function

$check_j$ performs this. Once again $check_j$ is a modification of an input-move but it must be noticed that in this case the communication matrix remains the same even after the value of C[i, j] is taken. That is to say, if

$$mvi_{j->i}\ (\lambda, m, \lambda, C, \varepsilon) = (\varepsilon, x' \leftarrow \mathrm{C}[j, i], m, \lambda, \Leftarrow C_{j,i}),\ \text{if C}[j, i] \neq @ \text{ and } cond$$

would have been utilised instead of

$$check_j\ (\lambda, m, \lambda, C, \varepsilon) = (\varepsilon, x' \leftarrow \mathrm{C}[i, j], m, \lambda, C),\ \text{if C}[i, j] \neq @ \text{ and } cond$$

then the resulting communicating matrix would have been $\Leftarrow C_{j,i}$ (i.e. an input variant of C, as in definition 4) and not C. Technically, this modification is indispensable because in the context of CSXMS-c, as has already been mentioned, the special symbol @ represents not only that a particular communication is not allowed but also that a machine has reached a final state. Thus, if C[i, j] = @ then it cannot longer be modified. For instance, the same reasoning applies for the *read_i* and *read_j* of the server.

In order to avoid unnecessary details, with particular explanations, for each extension or modification related to communication functions, from here on, all the communication functions that correspond to a input-move or an output-move as in definition 5 will be written using the names $mvi_{j->i}$ or $mvo_{i->j}$. For the rest of communication functions new names will be employed.

The function *request* followed by $mvo_{i->n+1}$, simply transmits to the server the request to send a message to machine P_j, that is $d_i \leftarrow j\mathrm{S}^+$. After that, P_i performs a loop reading the value in d_i until it is different from $j\mathrm{S}^+$ (function $whileS_j$). Once the server has changed the value, this is detected by the function end_loopS_j. Then it is evaluated (in *is_ok* and *otherwise*) whether the element in d_i corresponds to the symbol ↵, and if that is the case, the operation continues. Otherwise, the control in P_i is transferred again to the initial state q_0 where the macro-function is restarted.

[**when** *cond* =>] *j* **!** *val*

Original specification [34]		Specification for process *i* using the X-machine notation
f_1:	**if** *cond* & $C_{ij} \neq @$ **then** -	$check_i\ (x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[$i, j$] ≠ @ and *cond*
f_2:	$d_i \leftarrow j\mathrm{S}^+$	$requestS_j(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y' \leftarrow j\mathrm{S}^+, C)$ $mvo_{i->n+1}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{i, n+1} \Leftarrow y)$, if $y \neq \lambda$ and C[i, n+1] = λ
f_3:	**if** $d_i \neq j\mathrm{S}^+$ **then** -	$end\_loopS_j(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[$i, n$+1] ≠ $j\mathrm{S}^+$
f_4:	**if** $d_i = j\mathrm{S}^+$ **then** –	$whileS_j(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[$i, n$+1] = $j\mathrm{S}^+$
f_5:	**if** d_i = ↵ **then** –	$is\_ok(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[$i, n$+1] = ↵
f_6:	**if** $d_i \neq$ ↵ **then** -	$otherwise(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[$i, n$+1] ≠ ↵
f_7:	$C_{ij} \leftarrow val$	$value(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y' \leftarrow val, C)$ $mvo_{i->j}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{i, j} \Leftarrow y)$, if $y \neq \lambda$ and C[i, j] = λ
f_8:	**if** C_{ji} = $ **then** $C_{ji} \leftarrow \lambda$	$rec\_ack_i(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[$j, i$] = $ $mvi_{j->i}\ (\lambda, m, y, C, \varepsilon) = (\varepsilon, x' \leftarrow \mathrm{C}[j, i], m, y, \Leftarrow C_{j,i})$, if C[$j, i$] ≠ λ
f_9:	$d_i \leftarrow \lambda$	$mvi_{n+1->i}\ (\lambda, m, y, C, \varepsilon) = (\varepsilon, x' \leftarrow \mathrm{C}[n+1, i], m, y, \Leftarrow C_{n+1,i})$, if C[$n$+1, i] ≠ λ

Table 3

Note that if it happens that the mechanism reaches state q_5 the request has been granted (i.e. rendezvous), thus the current send can be executed. The function *value* assigns the value that will be transmitted to the output port and function $mvo_{i->j}$ writes it down in C[i, j], but on completion it must wait a signal (acknowledgement message) from P_j to say that the value has been successfully received. Function rec_ack_j actively awaits for the modification of cell C[j, i] and the receive of acknowledgement symbol $ (in function *is_$*), means that the macro-function has been executed properly and before its completion, it is necessary to clean the current cells C[j, i] and C[n+1, i] (in $mvi_{j->i}$ and $mvi_{n+1->i}$).

The complementary alternatives to the previous one have the form [**when** *cond* =>] *j* **?** *v* with the meaning: if *cond* is fulfilled, then local memory of P_i (in *v*) has to receive a value from machine P_j. The operation of this alternative, which is carried out by machine P_i is quite similar to the send counterpart as is shown in figure 13.b. Table 3 presents the specification of the output alternative in this form

Initially, machine P_i determines whether *cond* is true and whether the value in C[*j*, *i*] is different from @. This is computed by means of *check_it*. The functions *request* and $mvo_{i->n+1}$ are performed to request for authorisation to receive a message from machine P_j. For this a loop is executed until the value in d_i is different from jR^+ (in $whileR_j$), and then, after the value in d_i has been changed, if the new symbol is ↵ the macro-function continues. For any other new sequence, the control is returned to the initial state and the process is reinitiated.

From the comments made during the specification of the first alternative, it follows immediately that P_i gets to state q_5 when receive request has been granted (i.e. rendezvous), thus the actual receive operation can take part. Regardless of the value sent by P_j, function $mvi_{j->i}$ takes it and *get_value* actually places it into the memory. This will mean that an acknowledgement must be forwarded. The computation then proceeds by assigning the symbol \$ to the output port (in *put_$*) and by sending it (in $mvo_{i->j}$). It is straightforward to observe that if the contents of C[*i*, *j*] are λ or @ (in *go_ahead*) the acknowledgement to terminate has been accepted by the complementary machine, and this enables the execution of the macro-function by removing the current symbol in d_i (i.e. the operation $mvi_{n+1->i}$). Table 4 presents the list of functions corresponding to this form.

[**when** *cond* =>] *j* **?** *v*

Original specification [34]		Specification for process *i* using the X-machine notation
g_1:	**if** *cond* & $C_{ij} \neq @$ **then** -	*check_it* $(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[*j*, *i*] ≠ @ and *cond*
g_2:	$d_i \leftarrow jR^+$	$requestR_j(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y' \leftarrow jR^+, C)$ $mvo_{i->n+1}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{i, n+1} \Leftarrow y)$, if $y \neq \lambda$ and C[*i*, *n*+1] = λ
g_3:	**if** $d_i \neq jR^+$ **then** -	$end\_loopR_j(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[*i*, *n*+1] ≠ jR^+
g_4:	**if** $d_i = jR^+$ **then** -	$whileR_j(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[*i*, *n*+1] = jR^+
g_5:	**if** $d_i =$ ↵ **then** -	$is\_ok(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[*i*, *n*+1] = ↵
g_6:	**if** $d_i \neq$ ↵ **then** -	$otherwise(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[*i*, *n*+1] ≠ ↵
g_7:	**if** $C_{ji} \neq \lambda$ **then** $v \leftarrow C_{ji}$; $C_{ji} \leftarrow \lambda$	$mvi_{j->i}(\lambda, m, y, C, \varepsilon) = (\varepsilon, x' \leftarrow C[j, i], m, y, \Leftarrow C_{j,i})$, if C[*j*, *i*] ≠ λ $get\_value(x, m, y, C, \varepsilon) = (\varepsilon, x, m'.v \leftarrow x, y, C)$
g_8:	$C_{ij} \leftarrow$ \$	$put\_\$(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y' \leftarrow \$, C)$ $mvo_{i->j}(x, m, y, C, \varepsilon) = (\varepsilon, x, m, \lambda, C_{i, j} \Leftarrow y)$, if $y \neq \lambda$ and C[*i*, *j*] = λ
g_9:	**if** $C_{ij} = \lambda$ **or** $C_{ij} = @$ **then** -	*go-ahead* $(x, m, y, C, \varepsilon) = (\varepsilon, x, m, y, C)$, if C[*i*, *j*] = λ or C[*i*, *j*] = @
g_{10}:	$d_i \leftarrow \lambda$	$mvi_{n+1->i}(\lambda, m, y, C, \varepsilon) = (\varepsilon, x' \leftarrow C[n+1, i], m, y, \Leftarrow C_{n+1,i})$, if C[*n*+1, *i*] ≠ λ

Table 4

The **terminate** alternative in machine P_i is only applied when for all the other alternatives in the select construct $d_j = @$. The terminate alternative consists of updating d_i with @ and for all C[*i*, *j*] ← @ and for all C[*j*, *i*] ← @. In other words, machine P_i stops and this happens when P_i is unable to send or receive messages to other machines that had already finished their respective computations.

6.2 Properties of the CSXMS-c

In this section, the properties of the CSXMS-c are analysed with respect to the correctness of the implementation of the select construct. We begin by exhibiting the correctness of the message passing for the CSXMS-c following the same ideas of the first part of Theorem 8 in [34].

Lemma 11: The implementation of the message passing in the communicating states is correct.

Proof. Let t be the moment when the server has authorised a transmission between P_i and P_j. Without loss of generality, let us say that P_i is trying to send a message to P_j. As a result, both machines must be in their respective states q_5 in time t (rendezvous), also $d_i = d_j = \hookleftarrow$ and moreover C[i, j] = C[j, i] = λ. It is obvious then, that at time t the only enabled function is *value* which puts *val* into the output port of P_i, leaving it in state q_6.

Now, the transmission is possible through the application of $mvo_{i->j}$ and $mvi_{i->j}$ but since C[i, j] = λ the latter cannot be executed until the former is performed. Once $mvo_{i->j}$ has been computed C[i, j] ← *val* and P_i has to be in state q_7 where function *rec_ack* emerges, however, since C[j, i] = λ the application of *rec_ack* must be delayed until $mvo_{j->i}$ emerging from state q_8 of P_j has been executed. Hence, the order of execution is compulsory: $mvo_{i->j}$ of P_i followed by $mvi_{i->j}$, *get_value*, *put_$* and $mvo_{j->i}$ of P_j. In that moment P_j will be in state q_9 and C[j, i] = $. Following the same reasoning, function *go_ahead* of P_j must wait until a function removes the symbol $ from C[j, i], therefore, the complete sequence has to be as follows:

$mvo_{i->j}$, $mvi_{i->j}$, *get_value*, *put_$*, $mvo_{j->i}$, *rec_ack*, $mvi_{j->i}$

Leaving machines P_i and P_j in their respective states q_9 with C[i, j] = λ after executing $mvi_{i->j}$ and C[j, i] = λ after executing $mvi_{j->i}$. Because of this, it is not hard to see that the order in which the following assignments: d_i ← @ and d_j ← @ are done by the server is unimportant.

The remaining functions are $mvi_{n+1->i}$ for P_i and *go_ahead* and $mvi_{n+1->j}$ for P_j, where clearly the intention is to assign the λ symbol to d_i and d_j. If such assignment is simultaneous, the protocol successfully terminates given that these actions are disjoint. Alternatively, if one machine, let us say P_i (a symmetrical argument applies for P_j), executes first its assignment then there will be a moment when d_i = λ and $d_j = \hookleftarrow$ with C[i, j] = C[j, i] = λ. In this case, if any machine including P_i tries to establish a communication with P_j, such an operation will fail for the reason that $d_j = \hookleftarrow$. Therefore, any (new) communication with P_j, must be delayed until d_j = λ. From that, it is straightforward that in any case the message is transmitted and at the end $d_i = d_j$ = λ and C[i, j] = C[j, i] = λ.

□

Let us turn our attention to the correctness of the handling of the matrix components other than the server, and accordingly by the second part of theorem 8 of [34] it is possible to claim that:

Lemma 12: The handling of the common matrix C by the components P_i with $i \neq n + 1$ is correct.

Proof. By lemma 11, it follows that the handling of the matrix is done correctly during a message transmission. It remains to prove that P_i handles correctly the matrix before stopping. Suppose that P_i is trying to execute the **terminate** alternative, that is P_i is trying to assign the following:

d_i ← @, C[i, j] ← @ and C[j, i] ← @ $\forall$ $1 \leq j \leq n + 1$ and there are three cases:

1. If all the machines P_j that communicate with P_i have already finished then the assignations of P_i will not conflict with other machines.
2. If there is one machine P_j in state q_0 of any of the two select alternatives for send/receive a message to/from P_i and if it is the case that P_i has not modified yet C[i, j] or C[j, i] respectively. Then P_j is able to assign $i\mathrm{S}^+$ or $i\mathrm{R}^+$ to d_j reaching state q_3. Clearly, P_j will remain in state q_3 until the server counter is equal to j. For the reason that d_i in that moment is λ or @ then $i\mathrm{S}^-$ or $i\mathrm{R}^-$ will be assigned to d_j and so the control of P_j will be transferred to q_0 again. This process continues in this manner until eventually P_i modifies C[i, j] and C[j, i] implying that P_j will not try to communicate further with P_i.
3. If there is one machine P_j that has not completed yet a receive operation from P_i. Then by lemma 11 the sequence of functions has to be:

   $mvo_{i->j}$, $mvi_{i->j}$, *get_value*, *put_$*, $mvo_{j->i}$, *rec_ack*, $mvi_{j->i}$

   this implies that P_j can only be in state q_9. If C[j, i] ≠ @ then C[j, i] = λ and by lemma 11 therefore P_j will terminate its receiving operation. If C[j, i] = @ then P_j will reach state q_{10} (i.n *go_ahead*) and also it will complete the operation.

□

Corollary 1: The implementation of the select constructs in the communicating states is correct.

Proof. From lemmas 11 and 12.

□

It is worth illustrating the central ideas of lemma 11 by an abstract example.

Example 4: Suppose that P_i and P_j are respectively computing j **!** val and i **?** m, where m is part of the local memory of P_j. If the server has already given the permission for the operation then both machines are in state q_5 and the rest of the computation is as presented in table 5.

P_i state	P_i state	P_i memory	Enabled Functions	Function Executed	C[*i*, *j*]	C[*j*, *i*]	D_i	d_i
q_5	q_5	*Undefined*	*value*	*value*	λ	λ	↵	↵
q_6	q_5	*Undefined*	$mvo_{i->j}$	$mvo_{i->j}$	*val*	λ	↵	↵
q_7	q_5	*undefined*	$mvi_{i->j}$	$mvi_{i->j}$	λ	λ	↵	↵
q_7	q_6	*undefined*	*get_value*	*get_value*	λ	λ	↵	↵
q_7	q_7	*val*	*put_$*	*put_$*	λ	λ	↵	↵
q_7	q_8	*val*	$mvo_{j->i}$	$mvo_{j->1}$	λ	$	↵	↵
q_7	q_9	*val*	*rec_ack*	*rec_ack*	λ	$	↵	↵
q_8	q_9	*val*	$mvi_{j->i}$	$mvi_{j->1}$	λ	λ	↵	↵
q_9	q_9	*val*	$mvi_{n+1->i}$ *go_ahead* $mvi_{n+1->j}$	*go_ahead*	λ	λ	↵	↵
q_9	q_{10}	*val*	$mvi_{n+1->i}$ $mvi_{n+1->j}$	$mvi_{n+1->i}$	λ	λ	λ	↵
q_{10}	q_{10}	*val*	$mvi_{n+1->j}$	$mvi_{n+1->j}$	λ	λ	λ	λ
q_{10}	q_{11}	*val*			λ	λ	λ	λ
q_{10}	q_{10}	*val*	$mvi_{n+1->j}$	$mvi_{n+1->j}$	λ	λ	λ	λ

Table 5

■

7. The communicating (Eilenberg) multiple-streams X-machines systems (CM-SXMS)

The formalisms described above for building communicating systems point to the conception (specification) of a system as a whole, more than as a set of independent parts. An alternative methodology MSS was proposed in [35]. The approach is based on two steps:

1. Specify EXM independently of the target system or use existing specifications as components of the new system.
2. Determine the way in which these components communicate each other.

Following [35] the advantages of this approach are:

1. There is no need to start the specification of a system from scratch, because it is possible to re-use existing specifications.
2. Communication and specification can be considered separate and distinct activities in the development of communicating systems. In this way, the same tools (notations) can be employed for specifying both EXM and communicating systems.

In their paper, the notation that is employed corresponds to the XMDL, which is a proposed ASCII-based language for EXM. In general, XMDL is a non-positional notation based on tags, which are employed in defining EXM elements (e.g. states, memory, alphabets, functions, etc.), a more detailed discussion of XMDL can be found in [18, 19].

Let us return to the MSS, where the communication between components is done using several streams associated with them. In this methodology, the EXM have their own standard input and output stream, however, when components are integrated as a part of a large system more streams are included, we should call these the *communicating streams* or *c-streams*. In this way, every c-stream relates two machines in order to transmit messages, where a c-stream acts as output for the sender and as input for the receiver. To achieve this in [35], the semantics just described were incorporated into not only the state-transition diagrams usually employed to represent the next-state function of the EXM (as we will describe below), but also into the syntax of XMDL. So the XMDL is augmented by an annotation for specifying communication, where function names previously defined (specified) for the EXM are re-declared to *read from* or *write to*, in a different machine (model) subject to certain conditions.

Before going further in the formalisation of the MSS methodology, we shall develop a different technique from XMDL because our research is aimed towards the presentation of the diverse communicating approaches within similar notations, maintaining as many tight connections as possible by means of the mathematical nomenclature generally employed for stand-alone EXM. Consequently, we will use and adapt the MSXM suggested in [27] where it is demonstrated that this MSXM is equivalent to the SXM model.

7.1 The CM-SXMS model

First of all, following the example in [35] (i.e. queue system with multiple cashiers) for the MSS when a function $\phi \in \Phi$ is executed, only one input stream, and only one output stream are considered. Therefore, the MSXM needs to be restricted in order to fit with this assumption. In this section, we suggest how this can be done and how a composite communicating system can be constructed from these machines.

Definition 21: A CM-SXM Λ_k is a M-SXM $\Lambda_k^k = (\Sigma_1, \Sigma_2,\ldots, \Sigma_k, \Gamma_1, \Gamma_2,\ldots, \Gamma_k, Q, M, \Phi, F, I, T, m_0)$ such that

The type $\Phi : M \times \Sigma_1 \times \Sigma_2 \times\ldots\times \Sigma_k \rightarrow \Gamma_1 \times \Gamma_2 \times\ldots\times \Gamma_k \times M$ is defined by:

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

$\forall\, m \in M$ and $\exists\, h_j \in \Sigma_j$ $1 \le j \le k$ (i.e. $h_j \neq \varepsilon$) either:
$\phi(m, \varepsilon,\ldots, h_j,\ldots, \varepsilon) = \perp$ or
$\exists\, m' \in M$, $t_i \in \Gamma_i$ $1 \le i \le k$ (i.e. $t_i \neq \varepsilon$) such that t_i depends on m and h_j
$\forall\, g_1^*, g_2^*,\ldots, g_k^*$ such that $g_i^* \in \Gamma_i^*$ $1 \le i \le k$,
$\forall\, s_1^*, s_2^*,\ldots, s_j^*$ such that $s_i^* \in \Sigma_i^*$ $1 \le i \le j$,
$\phi(m, \varepsilon,\ldots, h_j,\ldots, \varepsilon) = (\varepsilon,\ldots, t_i, \ldots, \varepsilon, m')$

From definition 21 it becomes clear that a CM-SXM contains exactly the same number of input and output streams (i.e. k), and every function is defined over and affects one and only one input and output stream. In a further transformation, and in order to produce simpler specifications for the type of the CM-SXM, let us introduce a shortened notation:

for all $1 \le j \le k$, let $\pi i_j : (\Sigma_1 \cup \{\varepsilon\}) \times\ldots\times (\Sigma_k \cup \{\varepsilon\}) \rightarrow \Sigma_j$ and $\pi o_j : (\Gamma_1 \cup \{\varepsilon\}) \times\ldots\times (\Gamma_k \cup \{\varepsilon\}) \rightarrow \Gamma_j$ be two projection functions and let us say $\Sigma^T = \Sigma_1 \cup\ldots\cup \Sigma_k$, $\Gamma^T = \Gamma_1 \cup\ldots\cup \Gamma_k$ and $K = \{1, 2,\ldots, k\}$.
If $h = < h_1, h_2,\ldots, h_k > \in (\Sigma_1 \cup \{\varepsilon\}) \times\ldots\times (\Sigma_k \cup \{\varepsilon\})$ and $t = <t_1, t_2,\ldots, t_k> \in (\Gamma_1 \cup \{\varepsilon\}) \times\ldots\times (\Gamma_k \cup \{\varepsilon\})$ then it is possible to write;

$\Phi : M \times K \times \Sigma^T \rightarrow \Gamma^T \times K \times M$ where $\phi(m, j, \pi i_j(h)) = (\pi o_i(t), i, m')$

to refer to $\phi(m, \varepsilon,\ldots, h_j,\ldots, \varepsilon) = (\varepsilon,\ldots, t_i, \ldots, \varepsilon, m')$, since $\pi i_j(h) = h_j$ and $\pi o_i(t) = t_i$.

Example 5: Consider a CM-SXM Λ_7 where $\phi(m, \varepsilon, \varepsilon, \varepsilon, w, \varepsilon, \varepsilon, \varepsilon) = (\varepsilon, v, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon, m') \in \Phi$. Then it can be referred to as its simplified form $\phi(m, 4, w) = (v, 2, m')$, or in words 'reads the symbol w from the fourth input stream and places the symbol v at the end of the second output stream'.

■

Definition 22: A CM-SXMS with n components is a pair $G_n = (V, E)$ where $V(G_n)$ is a set of n CM-SXM of the form $\Lambda_n(i) = (\Sigma_1^i, \Sigma_2^i,\ldots, \Sigma_n^i, \Gamma_1^i, \Gamma_2^i,\ldots, \Gamma_n^i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{0i})$ and $E(G_n) \subset V(G_n) \times V(G_n)$ such that:

If $(\Lambda_n(i), \Lambda_n(j)) \in E(G_n)$ with $i \neq j$, this means that a connection is open between $\Lambda_n(i)$ and $\Lambda_n(j)$ where the *j-th* output communicating stream of $\Lambda_n(i)$ corresponds to the *i-th* input communicating stream of $\Lambda_n(j)$, in other words a c-stream and therefore $\Gamma_j^i = \Sigma_i^j$.

The *i-th* input and output streams of $\Lambda_n(i)$ are the standard streams (i.e. not c-streams) for the machine. It follows that Σ_i^i and Γ_i^i stand for the input and output alphabets and therefore $(\Lambda_n(i), \Lambda_n(i)) \notin E(G_n)$ and it is not necessary that $\Gamma_i^i = \Sigma_i^i$.

If $(\Lambda_n(i), \Lambda_n(j)) \notin E(G_n)$ then $\Lambda_n(i)$ and $\Lambda_n(j)$ do not communicate implying that the *j-th* output communicating stream of $\Lambda_n^n(i)$ and the *i-th* input communicating stream of $\Lambda_n(j)$ are always empty. To ensure this, $\neg\exists\ \phi(m, x, \pi i_x(h)) = (\pi o_j(t), j, m') \in \Phi_i$ and $\neg\exists\ \phi(m, i, \pi i_i(h)) = (\pi o_x(t), x, m') \in \Phi_j$ moreover $\Gamma_j^i = \Sigma_i^j = \varnothing$.

7.2 Specifying a process farm with a CM-SXMS

In this section, the specification of the farmer process system is carried out. Keeping in mind the methodological basis of the MSS, let us start with the specification of the components as SXM without regard to the target system.

The farmer process in this context is a SXM that copies each symbol from the input stream into the output stream. Therefore, the machine consists of a unique state q such that $q \in I$ and $q \in T$, also there is just one processing function $\phi(m, h) = (h, m) \in \Phi$ and $F(q, \phi) = q$. The reaper is also a one-state machine that stores in its memory a summation of all the symbols (i.e. integers) that have been received until a particular point in time, and this value is put into the output stream with each transition that is performed. That is, $Q = \{q\}$, $q \in I$, $q \in T$, $m_0 = 0$ with $\Phi = \{\phi\}$ where $\phi(m, h) = (m', m' \leftarrow m + h)$ and $F(q, \phi) = q$. The workers calculate the multiplication of every symbol received, and as in sections 3.3 and 4.2, let us say that each of these symbols is a pair (a_i, b_i). So the machine can be formalised as, $Q = \{q_0\}$, $q_0 \in I$, $q_0 \in T$, $\phi(m, h) = (h.a * h.b, m) \in \Phi$ and $F(q_0, \phi) = q_0$.

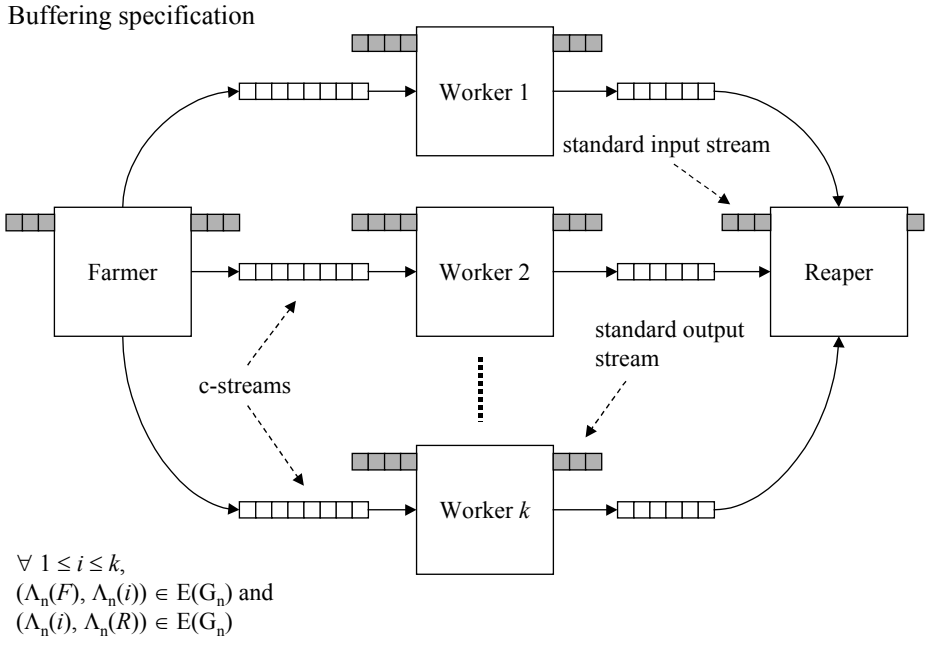


Figure 14

The second step in MSS consists of determining the way in which these SXM communicate. However, it is worth mentioning beforehand that the MSS methodology advocates the concept of several streams associated with each component to perform the transmission of messages. Hence, these streams have replaced the communication matrix of the other CSXMS approaches. A main implication has to do with the level of asynchrony permitted by this multiple-streams idea. Practically speaking, in the CSXMS a new message cannot be sent to a particular component if the previous message has not been read. By contrast, in the CM-SXMS,

since the media is now modelled as a stream, this admits in principle the transmission of several messages to the same component without regard to how many of them have been received.

The above situation allows modelling of the interactions farmer-worker and worker-reaper without concern to whether the receiver is ready or not to take a new message. In the CXM-system case, the Packet and Ready channels supported these interactions. For the CSXMS case the synchronisation-interaction was achieved by the way in which the communicating functions operated on the matrix. Nonetheless, such kind of interactions (i.e. acknowledgement, request for new tasks, etc.) can be specified in a CM-SXMS by establishing a two-way communication between the processes, together with the appropriate definition of the functions of the components. For the present example, let us specify the farmer process using both approaches and let us called them respectively the "buffering" and the "acknowledgement" specifications. As a convention, the standard streams are shaded to distinguish between them and the c-streams in the following figures.

The structure of the buffering system is presented in figure 14 where $n = k + 2$ and k is the number of worker processes in the system and let us enumerate $V(G_n) = \{\Lambda_n(1), \Lambda_n(2),\ldots, \Lambda_n(n)\}$ where $\Lambda_n(F)$ with $F = n - 1$ corresponds to the farmer, $\Lambda_n(R)$ with $R = n$ corresponds to the reaper process and $\Lambda_n(1),\ldots, \Lambda_n(k)$ are the workers. In this way the farmer $\Lambda_n(F)$ has a connection with the workers through its *1-st*, 2-*nd*,…, *k-th* output streams. Every worker interacts with the farmer using its *F-th* input stream and sends the results to the reaper via its *R-th* output stream. The reaper receives messages from the workers from its *1-st*, 2-*nd*,…, *k-th* input streams. Now let us specify the system assuming as in section 3.3 and 4.2 that a new message cannot be sent from the farmer to any of the workers if it has not requested the task explicitly

Let us suppose that the operation between a worker and the reaper is synchronised by a message that the latter explicitly sends to ask for a new partial result. The communicating structure of the system in this case is as in figure 15.

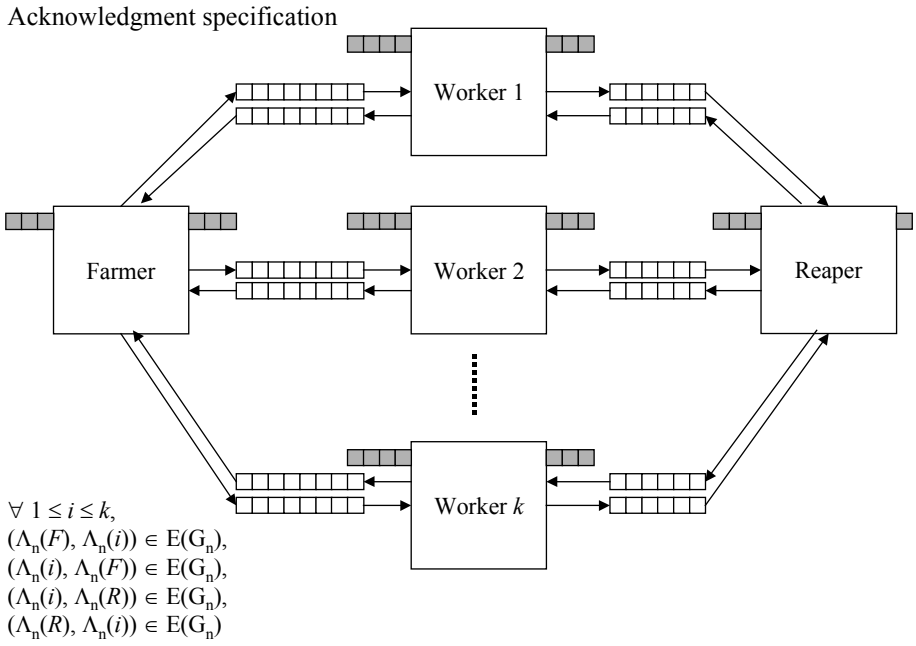


Figure 15

To complete the specification, it is necessary to adapt the stand-alone machines to operate as CM-SXM and perform the communication actions required by the system. Graphically, when a function acts over a c-stream, instead of a standard stream, a special annotation is directly joined to the edge that represents such a function. The connector point is drawn as a circle if the c-stream is acting as input or as a rectangle if the c-stream is acting as output. Additionally, the identification of the machine, with which the c-stream is shared, is included in the same annotation (e.g. $\Lambda_n(F)$).

a. farmer process.

For the case of the buffering specification, this construction is direct as it is illustrated in figure 16.a. As in the other examples, assume that all the elements of the vectors $v_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and $v_2 = \langle b_1, b_2, \ldots, b_n \rangle$ arrive at $\Lambda_n(F)$ (i.e. the farmer) in pairs of the form (a_i, b_i) and thereafter it distributes these pairs among the workers one after the other in round-robin. In order to do this, the memory is used as a counter from 0 to $k - 1$.

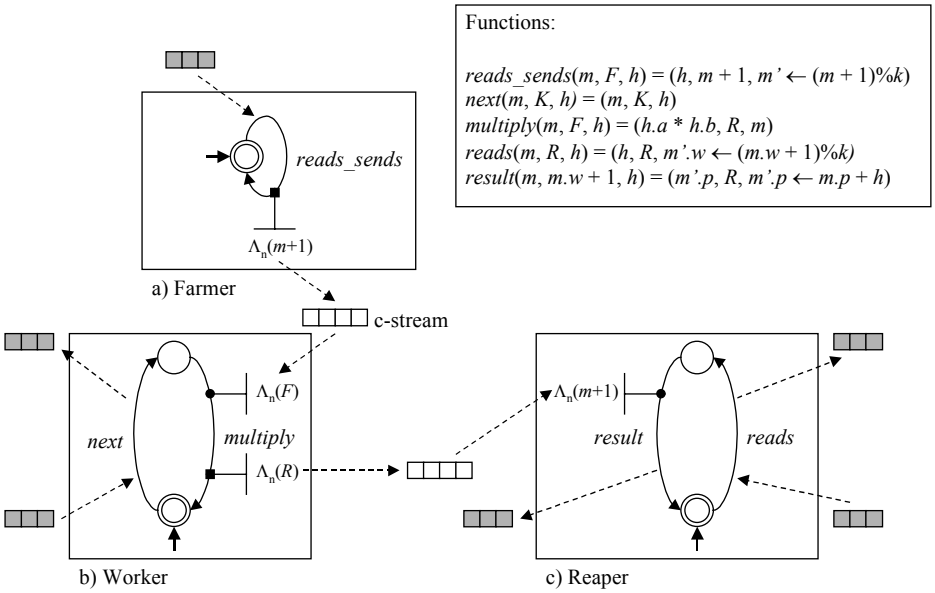


Figure 16

However, since the identifiers of the workers range from 1 to k, the value of this counter is increased by one every time a worker is chosen. Obviously, the initial memory value has to be zero.

On the other hand, the acknowledgement version of the farmer has two functions. The function *reads_pair* simply accepts a pair from the standard input stream and stores it in memory while the function *sends_pair* waits for the request for a new task from the worker and when it happens the pair in memory is then sent to the worker. Therefore, the memory contains two variables namely *m.w* and *m.p* the former is used as the counter for the round-robin loop and the latter stores the pair that will be sent. This is shown in figure 17.a, where the farmer is represented as a two-states machine to facilitate the visualisation, however it is easy to see that an equivalent one-state farmer can be drawn.

b. worker process

The worker for the buffering specification contains two functions and as in the previous case it is drawn as a two-states machine to facilitate the visualisation. The reason for having two functions is to ensure detectability (i.e. to avoid specifications with empty-operations) and to avoid situations, similar to the one described in section 3.2. In these situations is not possible to be certain if the termination condition, with respect to the inputs, has been reached. To be more precise, if all the input streams of a CM-SXM are empty then any function is undefined and the machine must stop, but what happens if a symbol arrives later in one of the input c-streams? Figure 16.b shows the specification of a worker.

The other version for the worker (figure 17.b) operates as follows. At the beginning, a symbol is read from the standard input stream and a request for a new task is sent to the farmer (in *ready*) then a pair is received from the worker and concatenated at the end of the standard output stream (in *receives*). Finally, in *multiply* the machine waits for an acknowledgement from the reaper and when it is received, the multiplication of the two elements of the last pair is transmitted to the reaper via the c-stream. Clearly, the same applies respect to the standard input stream that is it will contain the unary representation of u_i.

c. reaper process.

As in the worker's case, the standard input stream has to contain the unary representation of n and it must be noticed that both specifications are quite similar. Intuitively, in function *reads* a symbol is taken from the standard input stream and is written in the standard output stream in the first case (figure 16.c) or in the c-stream (figure 17.c) to indicate that the reaper is ready to receive a new message from the worker. At the same time, the counter memory variable is updated following the same type of computations as the farmer for the control of the round-robin loop.

In this way, the function *result* receives a partial result from a worker, computes the summation, stores this result in the memory accumulator, and concatenates it at the end of the standard output stream. Thus, the memory is organised as a duple where $m.p$ maintains the summation and $m.w$ is the counter and it is easily seen that $m_0.s = m_0.c = 0$.

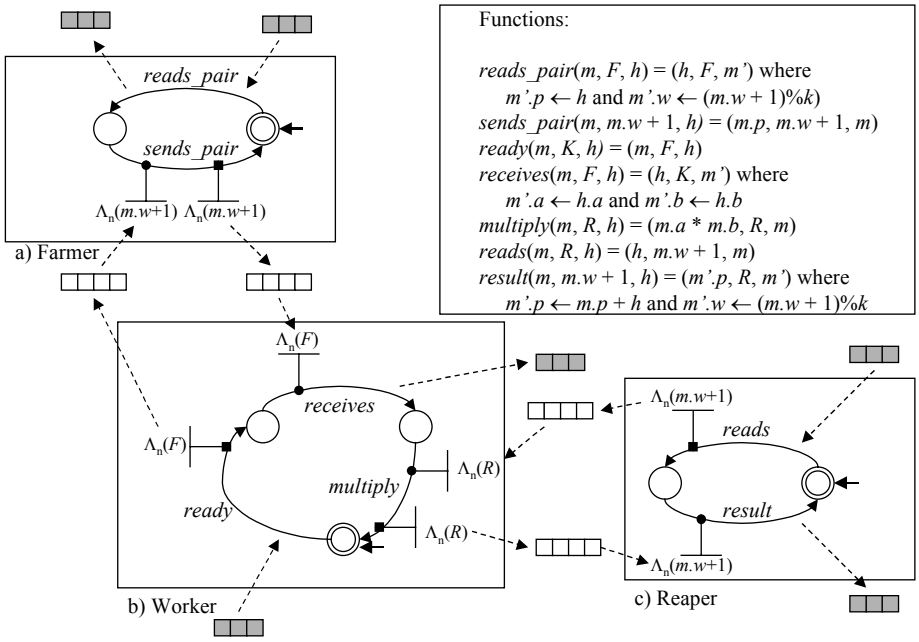


Figure 17

8. Conclusions

This report has summarised what has been archived so far in developing the communicating EXM models, and speculates on possible future research directions. It covers a wide range of these communicating models (all of them as far as we known) and their related theory, and attempts to fit them into a single general mathematical notation, similar to the one employed for the stand-alone EXM.

Several different proposal for providing communication capabilities for the EXM have been presented in the literature and it seems that by means of these models it is possible to specify distributed systems and algorithms. Additionally, from the testing perspective, it has been proved that the SXMT developed originally for SXM, can be applied to some of the communicating EXM systems.

The CXM-system extends the EXM model by means of including ports on it. The communication subsystem is represented by channels in such a way that each output port of one component might be connected to an input port of another machine. The particularity of the CXM-system in relation to the functional testing has been discussed in this document. In brief, the CXM-system has to be augmented or modified if the SXMT method will be used. This augmentation or modification related to allow the possibility of deriving the corresponding output of a given input (i.e. input-output relationship).

A promising approach that solves the above difficulty is the CSXMS, which represents the system's components as SXM and models the communication structure in terms of a matrix. It has been demonstrated that a CSXMS can be represented by an EXM that computes the same input-output relationship, and which can be constructed by an algorithmic procedure. Furthermore, the $CSXMS_n^T$ is an abstraction that guarantees that the machine resulting from this transformation procedure is an SXM, instead of a Sm-SXM.

It has been established that the SMXT finds all the faults if certain design-for-test conditions are fulfilled by the SXM, these conditions are completeness and output-distinguishability. An interesting property related to this is that the $CSXMS_n^T$ is output-distinguishable whenever all the components of the system are. For the completeness property, a subclass of CSXMS called the CSXMS-s has been identified such that whenever all the components are complete the entire system is input complete (relaxed variant of completeness). However, this is not a restriction for the applicability of the SXMT method, and therefore from a CSXMS-s its $CSXMS_n^T$ can be obtained and thus the system can be tested using this testing approach.

The CSXMS-c implements a means for using channels as the basic form for exchanging messages, even though the channels were originally used in the CXM-system, the CSXMS-c employs the concept in a slightly different way. It was proved that by introducing a special purpose machine called server, it is possible to achieve communication in a structured and synchronic way. The communication is structured because select construct appear in each communicating state, and the transmission is controlled by the server. In this way, the server either authorises or rejects the request from the components for passing messages; this decision depends on the configuration of the system and the operations of communication are done by a rendezvous-like synchronisation mechanism. Thus, to control the communication operations a macro-function or protocol is performed and it occurs among the machines that are trying to communicate and the server. The underlying structure of the CSXMS-c corresponds to the same communication matrix defined for the CSXMS and it was demonstrated the correct implementation of this approach. Particularly, the correctness of the message-passing has been shown and the correct handling of the matrix by the system's components.

A methodology called the MSS for building communicating systems from existing stand-alone SXM was analysed. The MSS differs to the other models in the sense that it is based on a bottom-up design style, that is to say, first the components has to be described and then the way in which these components communicate is specified. We approached MSS from a perspective supported in the original notation of the EXM, therefore, we presented and studied two new models the CM-SXM and the CM-SXMS. Put simply, a MSXM is an SXM with multiple inputs and outputs streams, and the machine operates over these streams in parallel with each function executed. The CM-SXM is a MSXM with the same number of input as output streams, and every function acts over one input and one output stream. In a CM-SXMS, the components are modelled by CM-SXM and the communication structure is given by shared streams, called here c-stream, in this way the output of a CM-SXM is the input of another. In order to ensure observable behaviour, all the CM-SXM in the system have a standard input and a standard output streams, in other words, streams that are not use for communication but for interact with the environment.

A fundamental issue in distributed systems has to do with the degree in which process activity takes place and how message-passing is synchronised. With respect to the latter, in a synchronous message-passing model the computation proceeds in such a way that a send/receive operation is co-ordinated and appears in single change of configuration of the system. At the other end is the asynchronous message-passing model in which there is no bound in the number of messages that can be send from a given component to another, and therefore these operations do not necessary take place in the same transition of the system.

As a remark, in the field of distributed computing the concepts of synchrony and asynchrony have a wide interpretation. For instance, the concepts of synchrony or asynchrony are also defined in terms of the amount of time elapsed between components' functions and in the bound (or no bound) on the time it can take for a message to be delivered. Distributed system can also be defined in terms of a semi-synchronous model in which these times (functions and message-delivery) can vary but they are bounded between two constant values [64].

Following the same ideas, it is possible to claim that a message-passing is semi-synchronous (or semi-asynchronous) if the number of messages that can be send without been received is bounded by a constant. Although synchronous approaches can be considered a special case of asynchronous models [65], and a significant step toward unifying these model has been taken [64]. The key point is that different architectures and design requirements have been resulted in these diverse approaches and a number of distributed algorithms have been presented and studied from these various perspectives. On the other hand, it seems that the CSXMS-c, CSXMS, and CM-SXMS fit respectively with the distributed computing models of synchronous, semi-synchronous, and asynchronous message-passing. Consequently, this diversity of communicating EXM models can represent an advantage in the sense that given a distributed solution for a problem, it is necessary to demonstrate its correctness. This has to do not only with some desired properties that the solution has to fulfil (i.e. safety and liveness) but also with the correct implementation of it (i.e. testing). Therefore, we believe that if the SXMT can be extended and applied to all of the communicating EXM system then it could be possible to test distributed algorithms with different message-passing structure, but this will require future work.

Areas for future research include the study of the similarity between the message-passing models and the corresponding systems of communicating EXM; the applicability of the SXMT approach to the CSXMS-c and CM-SXMS and the feasibility of the transformation from CM-CXMS to CSXMS or to SXM.

References

[1] S. Eilenberg. *Automata, Languages and Machines*, Vol. A, Academic press, N.Y. 1974.
[2] M. Holcombe. *X-Machines as basis for dynamic systems specification*, Software Engineering Journal, Vol. 3, No. 2, pp. 69-76, 1988.
[3] M. Stannett. *X-machines and the Halting Problem: Building a super-Turing machine*, Formal Aspects of Computing, Vol.2, pp. 331-341, 1990.
[4] G. Laycock. *The Theory and Practice of Specification-Based Software Testing*, Ph.D. Thesis, University of Sheffield, 1992.
[5] M. Fairtlough, M. Holcombe, F. Ipate, C. Jordan, G. Laycock and Z. Duan. *Using an X-machine to model a video cassette recorder*, Current issues in Electronic modelling, Vol. 3, pp. 141-151, 1995.
[6] F. Ipate. *Theory of X-machines and Applications in Specification and Testing*, Ph.D. Thesis, University of Sheffield, 1995.
[7] Z. Duan. *Modelling of Hybrid Systems*, Ph.D. Thesis, University of Sheffield, 1996.
[8] F. Ipate and M. Holcombe. *Another look at Computability*, Informatica, Vol. 20, pp. 359-372, 1996.
[9] K. Bogdanov, M. Fairtlough, M. Holcombe, F. Ipate and C. Jordan, *X-machine Specification and Refinement of Digital Devices*, Research Report CS-97-16, Department of Computer Science, University of Sheffield, 1997.
[10] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*, Springer Verlag Series on Applied Computing, 1998.
[11] F. Ipate. *Using Hybrid Machines for Specifying Hybrid Software Systems*, in Proceedings of the 4th International Symposium of Economic Informatics, Bucharest, pp. 679-686, Hungry, 1999.
[12] E. Berki. *Process Metamodels and Method Engineering in Software Process Improvement*, in the Proc. Of the BCS INSPIRE IV Conference: Training and Teaching for Understanding of Software Quality, ISBN 1-902505-36-0, University of North London, UK, Sep. 2000.
[13] T. Balanescu, M. Gheorghe, M. Holcombe. *Deterministic Stream X-machines based on grammar systems*, in Words, sequences, grammars, languages: where computer science, linguistics and mathematics meet, Vol. I, Ed. C. Martin-Vide & Mitrana, Kluwer, 2000.
[14] G. Eleftherakis. *Model Checking and X-machine Specification*, Technical Report CS-02/00, City College, Thessaloniki, Greece, 2000.
[15] M. Gheorghe. *Generalised Stream X-Machines and Cooperating Distributed Grammar Systems*, Formal Aspects of Computing, Vol. 12, No. 6, pp. 459-472, 2000.
[16] F. Ipate and M. Popescu. *A Z type Language for Specifying X-machines*, in Proceeding of CITTI, Constanta, Romania, pp. 82-88, 2000.
[17] P. Kefalas. *Modelling an Agent Reactive Architecture with X-machines*, Technical Report CS-01/00, City College, Thessaloniki, Greece, 2000.
[18] P. Kefalas. *X-machine Description Language: User Manual, version 1.6*, Technical Report CS-07/00, City College, Thessaloniki, Greece, 2000.
[19] P. Kefalas and E. Kapeti. *A design Language and Tool for X-machines Specifications*, in Advances in Informatics edited by D. I. Fotadis, S. D. Nikolopoulos, Word Scientific, pp. 134-145, 2000.
[20] P. Kefalas and A. Sotiriadou. *Transforming X-machines to Z Specifications*, Technical Report CS-06/oo, City College, Thessaloniki, Greece, 2000.
[21] S. Vanak. *X-Machines + L-Systems = XL-Systems*, International Conference on Computer Graphics 2000, Geneva, Switzerland, pp. 127-134, 2000.
[22] G. Eleftherakis and P. Kefalas. *Model Checking Safety-Critical Systems Specified as X-Machines*, to appear in Annals of Bucharest University.
[23] M. Stannett and T. Simons. *Complete Behavioural Testing of Object-Oriented Systems using CCS-Augmented X-machines*. Submitted to ECCOP 2002.
[24] J. Aguado, T. Balanescu, T. Cowling, M. Gheorghe, M. Holcombe and F. Ipate. *P Systems with Replicated Rewriting and Stream X-Machines (Eilenberg Machines)*, Fundamenta Informaticae, IOS Press, No. 49, pp. 1-17, 2001.
[25] G. Laycock. *Introductions to X-machines*. Research Report CS-93-13, Department of Computer Science, University of Sheffield, 1993.
[26] T. Balanescu, M. Gheorghe, M. Holcombe. *A subclass of stream X-machines with underlying distributed grammars*, Proceedings, Grammars Systems 2000, Ed. R Freund & A. Kelemenova, Silesian University at Opava, Czech Republic, ISBN 80-7248-067-7, pp. 93-112, 2000.

[27] J. Aguado and A. J. Cowling. *Foundations of the X-machine Theory for Testing*, Research Report CS-02-06, Department of Computer Science, University of Sheffield, 2002.
[28] J. Barnard, J. Whitworth and M. Woodward. *Communicating X-machines*, Journal of Information and Software Technology, Vol. 38, pp. 401-407, 1996.
[29] J. Barnard. *COMX: a design methodology using communicating X-machines*, Inform. Software Tech., Vol. 40(5-6), pp. 271-280, 1998.
[30] J. Barnard. *Object COMX: Methodology using communicating X-machine objects*, Journal of Object-Oriented Prog. Vol. 12, No. 7, pp. 12-17, 1999.
[31] T. Balanescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe and C. Vertan. *Communicating stream X-machines are no more than X-machines*, Journal of Universal Computer Science, Vol. 5, No. 9, pp 494-507, 1999.
[32] H. Georgescu and C Vertan. *A New Approach to Communicating Stream X-machines*, Journal of Universal Computer Science, Vol. 6, No 5, pp. 490-502, 2000.
[33] F. Ipate and M. Holcombe. *Testing Conditions for Communicating Stream X-machine*, Under consideration for publication in Formal Aspects of Computing
[34] A. J. Cowling, H. Georgescu and C. Vertan. *A Structured Way to use Channels for Communication in X-machines Systems*. Formal Aspects of Computing, Vol. 12, No. 6, pp. 485-500, 2000.
[35] P. Kefalas, G. Eleftherakis and E. Kehris. *Communicating X-Machines: A Practical Approach for Modular Specification of Large Systems*, Technical Report, CS-09-00, City College, Thessaloniki, Greece.
[36] S. Mauw and G.J. Veltink. *Algebraic Specification of Communication Protocols*. Cambridge Tracts in Theoretical Computer Science 36, 1993.
[37] M. Holcombe. *An integrated methodology for the formal specification, verification and testing of systems*, Proc. EuroSTAR 93, London, 1993.
[38] F. Ipate. *X-machine Based Testing*, in Proceedings of the 10-th International Conference on Control Systems and Computer Science, Vol. 2, pp. 262-272, Bucharest, 1995.
[39] M. Holcombe and F. Ipate. *Almost all Software Testing is Futile*. Research Report CS-95-03, Department of Computer Science, University of Sheffield, 1995.
[40] M. Holcombe, F. Ipate and A. Grondoudis. *Complete functional testing of safety critical systems*, Proc. IFAC Workshop on Emerging Control Technologies, pp. 343-358, Florida, 1995.
[41] M. Holcombe. *When testing is done*, Procc. EuroSTAR 97, Edinburgh, 1997.
[42] F. Ipate. *Is Software Testing Effective ?*, in Proceedings of the 3rd International Symposium of Economic informatics, Bucharest, pp. 173-179, 1997.
[43] F. Ipate and M. Holcombe. *An integration Testing method which is proved to find all faults*, Intern. J. Computer Math. Vol. 63, pp. 159-178, 1997.
[44] F. Ipate and M. Holcombe. *A method for refining and testing generalised machine specification*. Intern. J. Computer. Math., Vol. 68, pp. 197-219, 1998.
[45] F. Ipate and M. Holcombe. *Specification and testing using generalised machines: a presentation and a case study*, Software Testing, Verification and Reliability, Vol. 8, pp 61-81, 1998.
[46] S. Chambers. *Applying X-machines in the Retrospective Testing of Computer Software*, Ph.D. Thesis, University of Sheffield, 2000.
[47] T. Balanescu. *Generalised Stream X-Machines with Output Delimited Type*, Formal Aspects of Computing, Vol. 12, No. 6, pp. 473-484, 2000.
[48] A. Grondoudis. *X-machine Based Specification and Design for Testing of the CATV Protocol*, Ph.D. Thesis, University of Sheffield, 2000.
[49] M. Holcombe, T. Balanescu, M. Gheorghe and P. Radovici-Marculescu, *On testing generalized stream X-machines*, in Gh Paun (Ed), Recent topics in Mathematical Computational Linguistics, Romanian Academy Publishing House, pp. 130-141, 2000.
[50] F. Ipate. *A Theory of Testing for X-machines*, in Proceedings of CAIM 2000 (Confederation on Applied and Industrial Mathematics), Pitesti, Romania, 2000.
[51] E. Kehris, G. Eleftherakis and P. Kefalas, *Using X-machines to Model and Test Discrete Event Simulation Programs*, in Proceedings of the 4th World MultiConference on Circuits, Systems, Communications and Computers (CSCC), Athens, 2000.
[52] T.S. Chow. *Testing Software Design Modeled by Finite-State Machines*. IEEE Transactions on Software Engineering, Vol. 4, No. 3, pp. 178-187, 1978.
[53] S. Fujiwara, G. Von Bochmann, F. khendek, M. Amalou and A. Ghedamsi. *Test Selection Based on Finite State Models*, IEEE Transactions on Software Engineering, Vol. 17, No. 6, pp. 591-603, 1991.
[54] F. Ipate. *A method for testing non-deterministic X-machines that finds all faults*, to appear in Proc. CAIM 99, Pitesi, 1999.
[55] R. M. Hierons and M. Harman. *Testing Conformance to a Quasi-Non-Deterministic Stream X-Machine*, Formal Aspects of Computing, Vol 12, No. 6, pp. 423-442, 2000.

[56] F. Ipate and M. Holcombe. *Generating Test Sets from Non-deterministic Stream X-Machines*. Formal Aspects of Computing, Vol. 12, No. 6, pp. 443-458, 2000.
[57] F. Ipate and M. Holcombe. *Testing non-deterministic X-machines*,in Words, sequences, grammars, languages: where computer science, linguistics and mathematics meet, Vol. II, Ed. C. Martin-Vide & Mitrana, Kluwer, 2000.
[58] R. M. Hierons and M. Harman. *Testing conformance of a deterministic implementation against a non-deterministic stream X-machine*. Brunel University, October 30 2001.
[59] F. Ipate and M. Holcombe. *Generating Test Sequences from Non-deterministic Generalised Stream X-machines*, to appear in FACS, 2001.
[60] J. Aguado. *Transfer Report*, supervised by A. J. Cowling. University of Sheffield, Nov. 2000.
[61] J. Aguado and A. J. Cowling. *Design Models and the Complexity of the Testing Problem for Distributed Systems*, International Workshop on Semantic Foundations of Engineering Design Languages, SFEDL 2002, In conjunction with the 5-th European Joint Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 6-14, 2002.
[62] J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, H. F. Ledgard and J. Heliard. *Reference Manual for the ADA programming language: Proposed standard document*, Report AD-A090 709/7, Department of Defense, July 1980.
[63] R. N. Taylor. *A General-Purpose Algorithm for Analysing Concurrent Programs*, Communications of the ACM, Vol. 26, No 5, pp. 362-376, May 1983.
[64] M. Herlihy, S. Rajsbaum and M. R. Tuttle. *Unifying Synchronous and Asynchronous Message-Passing Models*, PODC 98, 1998.
[65] B. Charron-Bost, F. Mattern and G. Tel. *Synchronous, asynchronous, and causally ordeded communication*, Distributed Comput. 9, pp. 173-191, 1996.