

# Communicating X-Machines: from Theory to Practice

Petros Kefalas, George Eleftherakis and Evangelos Kehris

City Liberal Studies, Affiliated College of the University of Sheffield,  
Computer Science Department, 13 Tsimiski Str., 54624 Thessaloniki, Greece

**Abstract.** Formal modeling of complex systems is a non-trivial task, especially if a formal method does not facilitate separate development of the components of a system. This paper describes a methodology of building communicating X-machines from existing stand-alone X-machine models and presents the theory that drives this methodology. A X-machine is a formal method that resembles a finite state machine but can model non-trivial data structures. This is accomplished by incorporating a typed memory tuple into the model as well as transitions labeled with functions that operate on inputs and memory values. A set of X-machines can exchange messages with each other, thus building a communicating system model. However, existing communicating X-machines theories imply that the components of a communicating system should be built from scratch. We suggest that modeling of complex systems can be split into two separate and distinct activities: (a) the modeling of stand-alone X-machine components and (b) the description of the communication between these components. This approach is based on a different view of the theory of communicating X-machines and it leads towards disciplined, practical, and modular development. The proposed methodology is accompanied by an example, which demonstrates the use of communicating X-machines towards the modeling of large-scale systems.

## 1 Introduction

Formal modeling of complex systems can be a non-trivial task if the formal method used does not provide the appropriate means of abstraction or does not facilitate gradual development of the system model. In addition, as the complexity of the system increases, it becomes clear that it is necessary to break down the system model into several components that need to communicate with each other. There are mainly two ways to accomplish this: either (a) to build a communicating system from scratch in which models and communications between them will be inseparable, or (b) to model (or even to use off-the-shelf) separate components and then establish the communication between them. We believe that the latter is a more disciplined approach since modeling of components and modeling of communication are viewed as two distinct activities. The communicating X-machines is a formal method that facilitates this disciplined approach. In this paper, we use X-machines for modeling communicating systems.

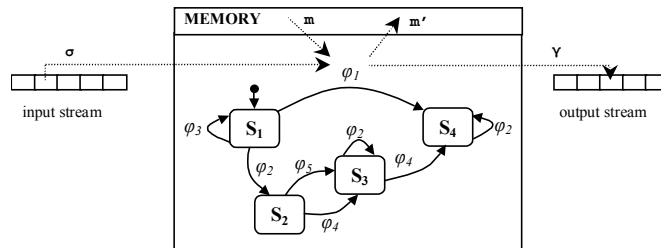
### 1.1 Definition of X-machine

A X-machine is a general computational machine introduced by Eilenberg [1] and extended by Holcombe [2], that resembles a Finite State Machine (FSM) but with two significant differences: (a) there is memory attached to the machine, and (b) the transitions are not labeled with simple inputs but with functions that operate on inputs and memory values. These differences allow the X-machines to be more expressive and flexible than the FSM.

The majority of formal languages facilitate the modeling of either the data processing or the control of a system [3]. X-machines can model non-trivial data structures as a typed memory tuple. X-machines employ a diagrammatic approach of modeling the control by extending the expressive power of the FSM. Therefore, X-machines are capable of modeling both the data and the control by integrating methods, which describe each of these aspects in the most appropriate way. Transitions between states are performed through the application of functions, which are written in a formal notation and model the processing of the data. Data is held in memory, which is attached to the X-machine. Functions receive input symbols and memory values, and produce output while modifying the memory values (Fig. 1). The machine, depending on the current state of control and the current values of the memory, consumes an input symbol from the input stream and determines the next state, the new memory state and the output symbol, which will be part of the output stream. The formal definition of a deterministic stream X-machine [4] is an 8-tuple:

$$M = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0), \text{ where:}$$

- $\Sigma, \Gamma$  is the input and output finite alphabet respectively,
- $Q$  is the finite set of states,
- $M$  is the (possibly) infinite set called memory,
- $\Phi$  is the type of the machine  $M$ , a finite set of partial functions  $\varphi$  that map an input and a memory state to an output and a new memory state,  $\varphi: \Sigma \times M \rightarrow \Gamma \times M$
- $F$  is the next state partial function that given a state and a function from the type  $\Phi$ , denotes the next state.  $F$  is often described as a transition state diagram,  $F: Q \times \Phi \rightarrow Q$
- $q_0$  and  $m_0$  are the initial state and memory respectively.



**Fig. 1.** An abstract example of a X-machine;  $\varphi_i$ : functions operating on inputs and memory,  $S_i$ : states.

X-Machines are more general than Turing Machines. They possess the computing power of Turing machines and, since they are more abstract, they are expressive enough to be closer to the implementation of a system. This feature makes them

particularly useful for modeling and also facilitates the implementation of various tools, which makes the development methodology built around X-machines more practical. In addition, X-machines not only provide a modeling formalism for a system but also offer a strategy to test the implementation against the model [5]. Ipate and Holcombe [4] presented a testing method, which is proved that it finds all faults in an implementation [6]. Furthermore, X-machines can be used as a core notation around which an integrated formal methodology of developing correct systems is built, ranging from model checking to testing [7].

## 1.2 Definition of Communicating X-machines

A Communicating X-machine model consists of several X-machines, which are able to exchange messages. These are normally viewed as inputs to some functions of a X-machine model, which in turn may affect the memory structure. A Communicating X-machine model can be generally defined as a tuple:

$$((M_i)_{i=1..n}, R), \text{ where}$$

- $M_i$  is the  $i^{\text{th}}$  X-machine that participates in the system, and
- $R$  is a communication relation between the  $n$  X-machines.

There are several approaches in order to formally define a communicating X-machine. Some of them deviate from the original definition of the X-machine  $M_i$ , which has the effect of not being able to reuse existing models. Also, different approaches define  $R$  in a different way, with the effect of achieving either synchronous or asynchronous communication.

In section 2, a review of the communicating X-machine theoretical approaches is presented together with the motivation and the theoretical background of our work. The practical contribution of this paper to communicating system modeling is analytically discussed in section 3 where our methodology as well as the appropriate notation is described. A concrete case study is presented in order to demonstrate the applicability of the methodology in section 4. Finally, in the last two sections the evaluation of the current approach is discussed and ideas for further work are given.

## 2 Theory of Communicating X-machines

### 2.1 The standard theoretical approach

A number of communicating X-machine approaches has been proposed [8,9,10]. A communicating stream X-machine with  $n$  components is a tuple [8]:

$$((XM_i)_{i=1..n}, CM, C_0) \text{ where:}$$

- $XM_i$  is the  $i^{\text{th}}$  X-machine of the system,
- $CM$  is a  $n \times n$  matrix, namely the Communication Matrix,
- $C_0$  is the initial communication matrix.

In this approach, a  $XM_i$  is different from a stand-alone X-machine definition  $M_i$ , since it utilizes an IN and an OUT port for communication (Fig.2). Both ports are linked to CM, which acts as the communication means between XMs. The CM cells contain “messages”, i.e. the cell  $(i,j)$  contains a “message” from  $XM_i$  to  $XM_j$ . The special value  $\lambda$  stands for “no message”. The “messages” can be of any type defined in all XM memories. In addition, there exist special kind of states and functions. The communicating functions emerge only from communication states, accept the empty symbol  $\varepsilon$  as input, and produce  $\varepsilon$  as output, while not affecting the memory. The communicating functions either read an element from CM and put it in the IN port, or write an element from the OUT port to the CM:

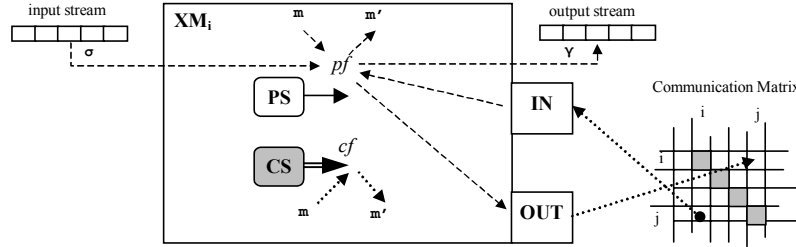
$$cf(\varepsilon, m, in, out, c) = (\varepsilon, m, in', out', c') \text{ where:}$$

$$m \in M, in, in' \in IN, out, out' \in OUT, c, c' \in CM$$

The communicating functions can write to the matrix only if the cell contains the special value  $\lambda$ . After the communicating functions read from CM, the cell is assigned the value  $\lambda$ . If a communication function is not applicable, it “waits” until it becomes applicable. The processing functions affect the contents of IN and OUT ports, emerge from processing states, and do not affect the communication matrix:

$$pf(\sigma, m, in, out) = (\gamma, m', in', out') \text{ where:}$$

$$\sigma \in \Sigma, m, m' \in M, in, in' \in IN, out, out' \in OUT, \gamma \in \Gamma$$



**Fig. 2.** An abstract example of a  $XMC_i$  with IN and OUT ports and the Communication Matrix

The above approach is sound and preserves the ability to generate a complete test set for the system, thus guarantying its correctness. A different methodology, namely COMX, of constructing communicating X-machines [10] also utilizes IN and OUT ports, which are described as a separate diagram. The methodology follows a top-down approach and the intention is to verify that certain properties of the communicating system are satisfied, such as reachability, boundness, deadlocks etc. A complex formal notation is used, which however is far from being standard in order to lead to the construction of appropriate tools. In addition, no effort is made to preserve the semantics of stand-alone X-machines, and therefore existing techniques for testing are unusable.

## 2.2 An alternative theoretical approach leading to practical development

The above approaches towards building communicating systems suffer one major drawback, i.e. a system should be conceived as a whole and not as a set of

independent components. As a consequence, one should start from scratch in order to specify a new component as part of the large system. In addition, specified components cannot be re-used as stand-alone X-machines or as components of other systems, since the formal definition of a X-machine M differs significantly from the definition of a XM. Moreover, the semantics of the functions affecting the communication matrix impose a limited asynchronous operation of a XM.

The alternative approach that we propose in this paper preserves to a great extent the standard theory described earlier. The only major difference is to abolish the communicating states and communicating functions and use an equivalent way to accomplish communication. In addition, the new proposal views the communicating system as a result of a sequence of operations that gradually transform a set of X-machines to a system model. These operations map conveniently to the practical methodology described in the following section.

Firstly, the X-machine Type ( $\mathcal{MT}$ ) is defined as a X-machine without an initial state and initial memory as the tuple:

$$\mathcal{MT} = (\Sigma, \Gamma, Q, M, \Phi, F)$$

A X-machine can be constructed through the application of the operator  $OP_{inst}$ .  $OP_{inst}: \mathcal{MT}_i \times (q_{0i}, m_{0i}) \rightarrow \mathcal{M}_i, \forall q_{0i} \in Q, m_{0i} \in M$ , which results in an instance of a  $\mathcal{MT}$ :

$$\mathcal{M} = \mathcal{MT} OP_{inst}(q_0, m_0)$$

A communicating X-machine component ( $\mathcal{XMC}$ ) is defined as the result of the following composition:

$$\mathcal{XMC}_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i) OP_{inst}(q_{0i}, m_{0i}) OP_{comm}(IS_i, OS_i, \Phi IS_i, \Phi OS_i)$$

where:

- $IS_i$  is a n-tuple that corresponds to n input streams, representing the input sources used for receiving messages from other  $\mathcal{XMC}$  ( $in_i$  is the standard input source of  $\mathcal{XMC}_i$ ):

$$IS_i = (is_1, is_2, \dots, is_i, \dots, is_n), \text{ and } is_j = \varepsilon \text{ (if no communication is required) or } is_j \subseteq \Sigma_i$$

- $OS_i$  is a tuple that corresponds to n output streams, representing the n output destinations used to send messages to n other  $\mathcal{XMC}$  ( $os_i$  is the standard output destination of  $\mathcal{XMC}_i$ ):

$$OS_i = (os_1, os_2, \dots, os_i, \dots, os_n), \text{ and } os_j = \varepsilon \text{ (if no communication is required) or } os_j \subseteq \Sigma_j$$

- $\Phi IS_i$  is an association of function  $\varphi_i \in \Phi_i$  and the input stream  $IS_i, \Phi IS_i: \varphi \leftrightarrow IS_i$
- $\Phi OS_i$  is an association of function  $\varphi_i \in \Phi_i$  and the output stream  $OS_i, \Phi OS_i: \varphi \leftrightarrow OS_i$

The application of the operator  $OP_{comm}: \mathcal{XMC}_i \times (IS_i, OS_i, \Phi IS_i, \Phi OS_i) \rightarrow \mathcal{XMC}_i$  has as a result a communicating X-machine component  $\mathcal{XMC}_i$  as a tuple:

$$\mathcal{XMC}_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi C_i, F_i, q_0, m_0, IS_i, OS_i), \text{ where:}$$

- $\Phi C_i$  is the new set of partial functions that read from either standard input or any other input stream and write to either the standard output or any other output stream.

Thus, the set consists of four different sets of functions, which combine any of the above possibilities:

$$\Phi C_i = SISO_i \cup SIOS_i \cup ISSO_i \cup ISOS_i$$

- $SISO_i$  is the set of functions  $\varphi$  that read from standard input stream ( $is_i$ ) and write to standard output stream ( $os_i$ ):

$$SISO_i = \{(is_i, m) \rightarrow (os_i, m) \mid \varphi_i = (\sigma, m) \rightarrow (\gamma, m) \in \Phi_i \wedge \varphi_i \notin \text{dom}(IS_i) \wedge \varphi_i \notin \text{dom}(OS_i)\}$$

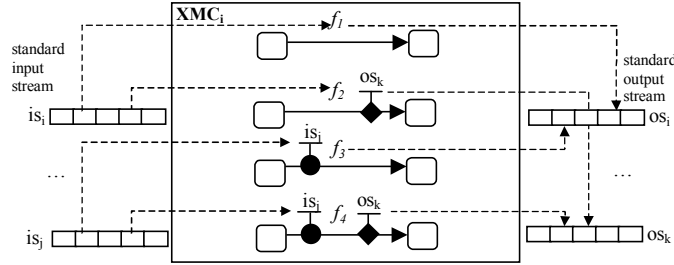
- $SIOS_i$  is the set of functions  $\varphi$  that read from standard input stream ( $is_i$ ) and write to the  $j^{\text{th}}$  output stream ( $os_j$ ):  
 $SIOS_i = \{(is_i, m) \rightarrow (os_j, m) \mid \varphi_i = (\sigma, m) \rightarrow (\gamma, m) \in \Phi_i \wedge \varphi_i \notin \text{dom}(IS_i) \wedge (\varphi_i \rightarrow os_j) \in OS_j\}$
- $ISSO_i$  is the set of functions  $\varphi$  that read from the  $j^{\text{th}}$  input stream ( $is_j$ ) and write to the standard output stream ( $os_i$ )  
 $ISSO_i = \{(is_j, m) \rightarrow (os_i, m) \mid \varphi_i = (\sigma, m) \rightarrow (\gamma, m) \in \Phi_i \wedge (\varphi_i \rightarrow is_j) \in IS_i \wedge \varphi_i \notin \text{dom}(OS_i)\}$
- $ISOS_i$  is the set of functions  $\varphi$  that read from the  $j^{\text{th}}$  input stream ( $is_j$ ) and write to the  $k^{\text{th}}$  output stream ( $os_k$ ):  
 $ISOS_i = \{(is_j, m) \rightarrow (os_k, m) \mid \varphi_i = (\sigma, m) \rightarrow (\gamma, m) \in \Phi_i \wedge (\varphi_i \rightarrow is_j) \in IS_i \wedge (\varphi_i \rightarrow os_k) \in OS_k\}$

Finally, the communicating X-machine is defined as a tuple of  $n$   $\mathcal{XMC}$  as follows:

$$CXM = (\mathcal{XMC}_1, \mathcal{XMC}_2, \dots, \mathcal{XMC}_n), \text{ with}$$

- $\Sigma_1 \cup \Sigma_2 \cup \dots \cup \Sigma_n = (os_{11} \cup os_{12} \cup \dots \cup os_{1n}) \cup \dots \cup (os_{n1} \cup os_{n2} \cup \dots \cup os_{nn})$ , and
- $\Gamma_1 \cup \Gamma_2 \cup \dots \cup \Gamma_n = (is_{11} \cup is_{12} \cup \dots \cup is_{1n}) \cup \dots \cup (is_{n1} \cup is_{n2} \cup \dots \cup is_{nn})$

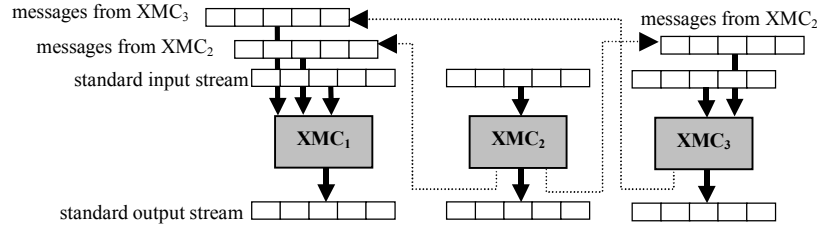
Fig.3 depicts the four different kinds of functions that may exist in a communicating X-machine component. For exposition reasons, we use a special graphical notation, namely the *solid circle* and the *solid diamond*. If a solid circle appears on a transition function, this function accepts input from the communicating steam instead of the standard input stream. If a solid diamond appears on a transition function, this function may write to a communicating input stream of another X-machine.



**Fig. 3.** An abstract example of a  $\mathcal{XMC}_i$  with input and output streams and functions that receive input and produce output in any possible combination of sources and destinations.

In our approach, we have replaced the communication matrix by several input streams associated with each X-machine component. Although, this may look only as a different conceptual view of the same entity, it will serve both exposition purposes as well as asynchronous operation of the individual machines. X-machines have their own standard input stream but when they are used as components of a large-scale system more streams may be added whenever it is necessary. The number of streams associated with one  $\mathcal{XMC}$  depends on the number of other  $\mathcal{XMC}$ s, from which it receives messages (Fig. 4).

The replacement of the communicating states and communicating functions with the same functions that belong to the type  $\Phi$  but with the option to read and write to different input and output streams, facilitate the practical development of communicating X-machine models.



**Fig. 4.** Three communicating X-machine components XMC<sub>1</sub>, XMC<sub>2</sub>, and XMC<sub>3</sub> and the resulting communicating system where XMC<sub>2</sub> communicates with XMC<sub>1</sub> and XMC<sub>3</sub>, while XMC<sub>3</sub> communicates with XMC<sub>1</sub>.

### 3 Practical Modeling of Communicating X-Machines

The alternative theoretical approach presented earlier leads towards a methodology of developing large-scale communicating systems. Since the communicating X-machine model is viewed as the composition of X-machine type with the initial memory and an initial state as well as with a set of input/output streams and associations of these streams to functions, the development of a model can be mapped into three distinct actions: (a) develop X-machine type models independently of the target system, or use existing models as they are, as components, (b) create X-machine instances of those types and (c) determine the way in which the independent instance models communicate. Optionally, a fourth step may follow, which extends the model of a communicating system in order to provide additional functionality.

The approach has several advantages for the developer who: (a) does not need to model a communicating system from scratch, (b) can re-use existing models, (c) can consider modeling and communication as two separate distinct activities in the development of a communicating system, and (d) can use existing tools for both stand-alone and communicating X-machines.

In addition, for a formal method to be practical, it needs certain tools that will facilitate modeling. A prerequisite for those tools is to use a standard notation to describe models in that formal language, other than any ad-hoc mathematical notation. The formal definitions of the X-machines can be presented using the notation of X-machine Description Language [11], which is intended to be an ASCII-based interchange language between X-machine tools [12,13]. Briefly, XMDL is a non-positional notation based on tags, used for the declaration of X-machine parts, e.g. types, set of states, memory, input and output symbols, functions etc. The functions take two parameter tuples, i.e. an input symbol and a memory value, and return two new parameter tuples, i.e. an output and a new memory value. A function may be applicable under conditions (*if-then*) or unconditionally. Variables are denoted by the prefix ?. The informative *where* in combination with the operator <- is used to describe operations on memory values. The functions are of the form:

```

#fun <function name> ( <input tuple> , <memory tuple> ) =
if <condition expression> then
    ( <output tuple>, <memory tuple> )
where    <informative expression>.

```

XMDL has been enriched with syntax that provide the ability to define instances of X-machine types. The syntax is the following:

```

#model <instance name> instance_of <model name>
[with:
#init_state = <instance initial state>.
#init_memory = <instance initial memory tuple>].

```

Finally, XMDL provides syntax to express the solid circle and the solid diamond that are attached to the functions of the communicating machine and denote input and output streams respectively. In order to incorporate the above semantics of communication, the syntax of XMDL is enhanced by the following annotation:

```

#communication of <model name>:
[<function name> reads from <model name>
where <expression> from (memory|input|output) <tuple>]'
[<function name> writes <message> to <model name>
where <expression> from (memory|input|output) <tuple>]'.

```

One can imagine that XMDL code for X-machine types are kept in different files from XMDL code for X-machine instances and communication definitions. All these files may be compiled into one in order to produce the communicating X-machine model in XMDL which can then be used for various reasons, e.g. testing, model checking etc. by other tools.

## 4 Case Study: Building a Communicating System

This section presents a case study of developing a communicating system based on the methodology described above. The notation used to define X-machine components is XMDL. The description of the system to be modeled as a communicating system is as follows: a lift with a finite capacity operates in a five-floor building by serving people that want to move from one floor of the building to another. The lift records and satisfies all the requests generated by people who are either inside the lift, or outside it: people outside the lift waiting on a floor call the lift by pressing the call button that is located at their floor, while people inside the lift determine their destination floor e.g. by pressing the appropriate button. The lift is moving among the floors to satisfy the generated requests. When the lift reaches a floor for which a request has been generated, it stops so that people may enter or leave the lift: people in the lift destined to that floor exit the lift while people waiting on that floor for the lift to arrive may enter the lift (provided the lift capacity is respected). When all possible loading and unloading of the lift is over, the lift may start moving to a new floor (if it carries people) or it may remain idle (if it is empty). It is assumed that people waiting for the lift to arrive on a floor, form a queue.

#### 4.1 Modeling of X-machine Type

In the above system, there are three types of X-machines to be modeled: the queue of people waiting at a floor, a call button in front of the lift door at each floor and the lift itself. Assuming that there is no limit in the number of people waiting at a floor, the queue is modeled as a X-machine with two states: *empty* and *filling* (Fig.5). The memory of the X-machine holds the actual sequence of persons waiting at a floor. Functions are activated by input tuples of the form  $(event, person)$ , where *event* is the arrival or departure of a person in the queue. In this case, *PERSON* is a basic type, i.e. anything, without any further reference to its implementation.

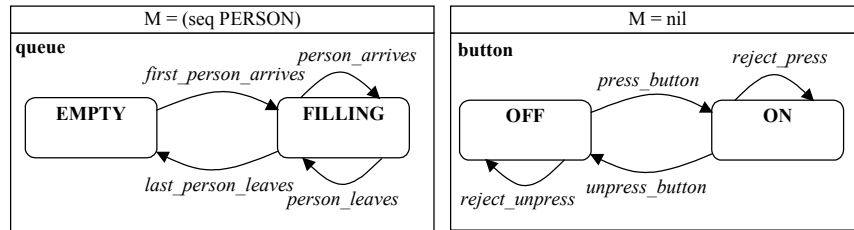


Fig. 5. The state transition diagrams of the queue and button X-machines.

The following is the XMDL listing for the specification of queue:

```
#model queue.
#basic_types = [PERSON].
#type messages = {FirstPersonInQueue, PersonInQueue, PersonLeft,
LastPersonLeft}.
#type event = {arrival, departure}.
#states = {empty, filling}.
#input (event, PERSON).
#output (messages, PERSON).
#memory (sequence_of PERSON).
#fun first_person_arrives((arrival, ?p), (nil)) =
if ?p belongs PERSON then
  ((FirstPersonInQueue, ?p), (<?p>)).
#fun person_arrives((arrival, ?p), (?queue)) =
if ?p belongs PERSON then
  ((PersonInQueue, ?p), (<?new_queue>))
where
  ?new_queue <- ?p addatendof ?queue.
#fun person_leaves((departure, ?p), (<?p::?queue>)) =
if ?queue /= nil then
  ((PersonLeft, ?p), (<?queue>)).
#fun last_person_leaves((departure, ?p), (<?p>)) =
  ((LastPersonLeft, ?p), (nil)).
#transition(empty, first_person_arrives)=filling.
#transition(filling, person_arrives)=filling.
#transition(filling, person_leaves)=filling.
#transition(filling, last_person_leaves)=empty.
```

Accordingly, the button is a X-machine with no memory, i.e. a finite state machine, with two states *off* and *on* (Fig.5):

```

#model button.
#type messages = {Pressed, NotPressed, AlreadyPressed,
AlreadyNotPressed}.
#type action = {press, unpress}.
#states = {on, off}.
#memory nil.
#input (action).
#output (messages).
#fun press_button((press), nil) = ((Pressed),nil).
#fun unpress_button((unpress),nil)=((NotPressed),nil).
#fun reject_press((press),nil)=((AlreadyPressed),nil).
#fun reject_unpress((unpress),nil)=((AlreadyNotPressed), nil).

```

Finally, the model lift is slightly more complicated (Fig.6). The memory of the X-machine consists of the floor at which the lift is currently at, the set of floors, which are recorded as destinations following requests, the set of people that are in it, the maximum floor number that the lift can serve and the total capacity in persons that the lift can bear:

```

#basic_types = [INDIVIDUAL].
#type floor = natural0.
#type capacity = natural.
#type destinations = set_of floor.
#type people = set_of INDIVIDUAL.
#memory (floor, destinations, people, floor, capacity).

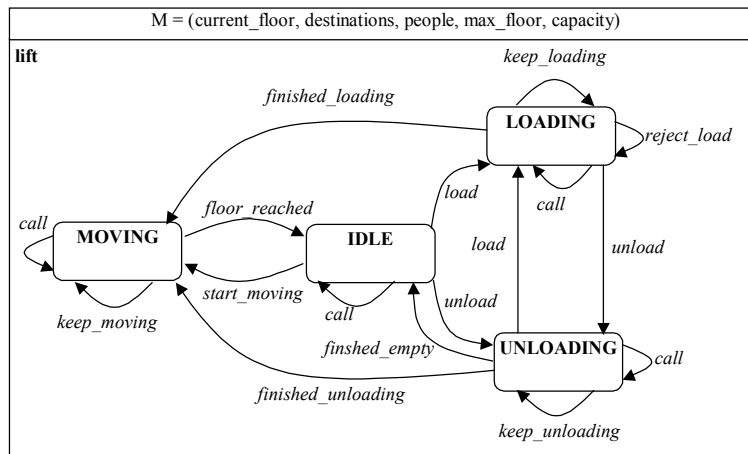
```

The lift can exist in four states:

```

#states = {moving, idle, loading, unloading}.

```



**Fig. 6.** The state transition diagram of the lift X-machine.

The input that the functions receive are defined as tuples in which the first element denotes the action to be performed while the second element is either a floor or a person making a request. The output is defined accordingly:

```

#type messages = {CallRequested, LiftAtFloor, LiftMoving,
LoadingPeople, UnloadingPeople, LiftPassedFloor, LiftFull}.

```

```

#type action = {get_in, get_out, request, continue}.
#input (action, floor union INDIVIDUAL).
#output (messages, floor union INDIVIDUAL).

```

While in any state, the lift can accept a request, either from a person within the lift or a person in any floor. The request triggers the function call, which is responsible to update the memory:

```

#fun call( (request, ?f), (?cf, ?d, ?p, ?mf, ?c)) =
if ?f belongs floor and ?f =< ?mf then
  ((CallRequested,?f), (?cf, ?nd, ?p, ?mf, ?c))
where
  ?nd <- ?f addsetelement ?d.

```

The rest of the functions are defined accordingly. Briefly, *load* and *keep\_loading* are triggered when an individual enters the lift, while *reject\_load* is triggered when the maximum capacity is reached. The functions *unload* and *keep\_unloading* are counterparts of the above when individuals exit the lift. *Finished\_loading* and *finished\_unloading* mark the start of the lift's move to its recorded destinations. When a floor that is in its destinations is reached the lift becomes stationary (*idle*) in this floor:

```

#fun floor_reached( (continue, ?f), (?cf, ?d, ?p, ?mf, ?c)) =
if ?f belongs floor and ?f belongs ?d and ?f =< ?mf then
  ((LiftAtFloor, ?f), (?f, ?nd, ?p, ?mf, ?c))
where
  ?nd <- ?f delsetelement ?d.

```

At this stage, the three independent models of the components of the systems are developed as if they were stand-alone X-machines, without initial state and initial memory.

## 4.2 Creation of X-Machine Instances

The system would require components, which are X-machine instances of the previously defined X-machine types. Assuming that the overall system consists of 5 floors and one lift that serves all five plus the ground floor (number 0), this is achieved through the following declarations:

```

#model queue0 instance_of queue with:
#init_state = {empty}
#init_memory = (nil).
#model queue1 instance_of queue with:
. . .
#model button_floor0 instance_of button with:
#init_state = {off}
#model button_floor1 instance_of button with:
. . .
#model my_lift instance_of lift with:
#init_state = {idle}
#init_memory = (0,nil,nil,10,5).

```

### 4.3 Modeling of the Communicating System

The communicating system consists of six queues (one for each floor), six buttons (one for each floor) and one lift (Fig.7). The communication between the components should be established so that when any person who arrives in the queue presses the button, which in turn causes a request to be recorded in the lift destinations. For example, the communication of the queue and the button at ground floor is established as follows (Fig.8):

```
#communication of queue0:
first_person_arrives writes (press) to button_floor0.
person_arrives writes (press) to button_floor0.

#communication of button_floor0:
press_button reads from queue0.
reject_press reads from queue0.
```

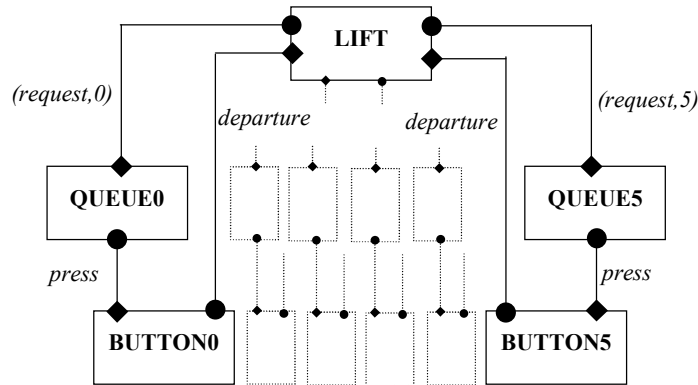
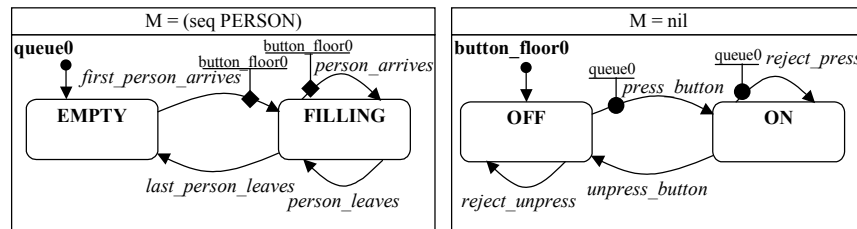


Fig. 7. The overall communicating X-machine system with the messages exchanged. The solid circle should be read as “from” while the solid diamond as “to”.

In addition, the communication of the button at ground floor and the lift is established with the following XMDL code:

```
#communication of button_floor0:
press_button writes (request,0) to my_lift.

#communication of my_lift:
call reads from button_floor0.
```



**Fig. 8.** The state transition diagrams for the communication between a queue and its related button.

On the other hand, when an individual enters the lift, the function *load* and *keep\_loading* should notify the functions *person\_leaves* and *last\_person\_leaves* of the model queue in order to remove the persons from the waiting queue. For example, this communication between the lift and the queue at the fifth floor is decoded as follows:

```
#communication of my_lift:
load writes (departure) to queue5
where ?f==5 from memory (?f,_,_,_,_).
keep_loading writes (departure) to queue5
where ?f==5 from memory (?f,_,_,_,_).

#communication of queue5:
person_leaves reads from my_lift.
last_person_leaves reads from my_lift.
```

Finally, the functions *finish\_loading* and *finish\_unloading* that are triggered in order to move the lift to the rest of the destinations, should notify the button of the specific floor to turn off:

```
#communication of my_lift:
finish_loading write (unpress) to button_floor4
where ?f==5 from memory (?f,_,_,_,_).
finish_unloading write (unpress) to button_floor4
where ?f==5 from memory (?f,_,_,_,_).
start_moving write (unpress) to button_floor4
where ?f==5 from memory (?f,_,_,_,_).

#communication of button_floor5:
unpress_button reads from my_lift.
reject_unpress reads from my_lift.
```

The communication, which is declared separately from the X-machine types and the X-machine instances, completes the overall description of the communicating system.

#### 4.4 Extending the Communicating System

Developing larger models as communicating systems from existing building blocks implies the need for some more features, which can be included in communicating X-machines. For example, if there are two lifts that serve the five-floor building, then there must be a X-machine created as an instance of the model lift and an additional X-machine that performs the scheduling (Fig.9). In the approaches presented by other researchers [8,9,10], this is not easily done. In the current approach, re-building the system includes only the modification of the communication part and the new specification of the scheduler. The simple scheduler that is presented below allocates a task to a device with the fewer tasks, only if the task is not recorded to any of the two devices (Fig. 10).

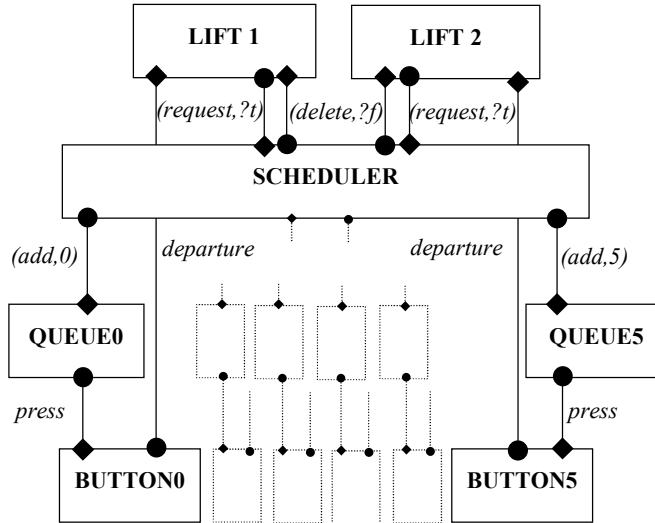


Fig. 9. The overall communicating X-machine system.

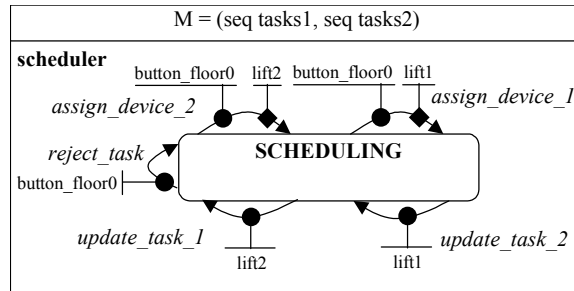


Fig.10. The transition diagram of the scheduler specification as a part of a communicating system

The complete model of the scheduler in XMDL can be found in [14]. The model can either be created from scratch, or it may already exist as a component of some other system. The scheduler is general and does not refer to any of the other two X-machines. However, the communication part is system specific:

```
#communication of scheduler:
reject_task reads from button_floor0.
.
.
.
assign_device1 reads from button_floor0.
.
.
.
assign_device2 reads from button_floor0.
.
.
.
update_task1 reads from lift1.
update_task2 reads from lift2.
```

```
assign_device1 writes (request,?t) to lift1
where ?t from input (_,?t)
assign_device2 writes (request,?t) to lift2.
where ?t from input (_,?t).
```

The communication part of the queues and the buttons should change accordingly. The kind of synchronization described above, also appears when buffering or synchronization is required. For example, if a function requires a n-tuple as input, then, assuming that every element of the n-tuple is produced by other machines, a new buffer X-machine should be specified in order to construct the n-tuple and then pass for consumption. In such cases, one can imagine “ready-made” generic X-machines that would act as synchronization interfaces or buffers between other machines in a communicating system.

## 5 Evaluation

In comparison with methodologies described elsewhere [8,9,10] the current approach is:

- *Practical*: one does not need different tools or a different notation, other than XMDL, for stand-alone as well as communicating systems.
- *Modular*: it leads towards component-based communicating X-machine models. Communicating systems are built from stand-alone X-machines. Parts of communicating systems can be re-used by simply changing the communication part.
- *Asynchronous*: X-machines are independent of each other and there is no synchronization imposed, as in reading and writing in communication matrix. Synchronisation is a property, which can be achieved through generic X-machines, if it is required.
- *Disciplined*: modeling and communication are regarded as two separate activities.
- *Sound*: the theoretical framework developed is equivalent to existing theoretical models of X-machines.
- *General*: it subsumes the existing approaches.

The way of describing the communication with annotations does not retract anything from the theoretical model containing processing as well as communicating functions and states. In fact, the X-machine models annotated in the way described above can be transformed into X-machines containing the two kinds of functions and states. A function that reads input from its own communication input stream, can be viewed as a communicating state followed by a communication function that reads the matrix and changes the IN port. If the function writes a message to another machine’s communication input stream, can be viewed as a processing state that writes to the OUT port, followed by a communicating state which in turn is followed by a communication function that writes to the matrix. This subsumption guarantees that the properties proved in [8] are also valid for the current approach.

## 6 Conclusions

We have presented a methodology for building communicating systems out of existing stand-alone X-machines. The theoretical framework described facilitates the practical development of communicating X-machine models. This is because the software engineer can separately specify the components and then describe the way in which these components communicate. This allows a disciplined development of large-scale systems. Also, X-machine models can be re-used in other systems, since one needs to change only the communication part. The major advantage is that the methodology also lends itself to modular testing and model checking strategies in which X-machines are individually tested and verified as components while communication is tested and verified separately.

We are currently applying the methodology for developing formal models of multi-agent systems [15, 16]. It is found that by using communicating X-machines, we can formally model the behaviour of reactive agents as well as their cooperation. Future work will include the implementation of communicating systems on top of the already existing tools [12,13] and an enhanced theoretical framework that will provide the ability to describe types of communication in accordance to types of models and also a more effective way to adapt off-the-shelf X-machine models that require minor changes in their specifications. Finally, a methodology for deriving a complete test set for communicating systems as well as a model checking strategy will be investigated [7].

## References

1. Eilenberg S.: Automata Machines and Languages, Vol. A, Academic Press, 1974.
2. Holcombe M.: X-machines as a basis for dynamic system specification, Software Engineering Journal, Vol.3, No.2 (1988) 69-76
3. Clarke E., Wing J. M.: Formal Methods: State of the Art and Future Directions, ACM Computing Surveys, Vol.28, No.4, (1996) 626-643
4. Ipate F. and Holcombe M.: Specification and testing using generalised machines: a presentation and a case study, Software Testing, Verification and Reliability, Vol.8 (1998) 61-81
5. Kehris E., Eleftherakis G., and Kefalas P.: Using X-machines to Model and Test Discrete Event Simulation Programs, In Systems and Control: Theory and Applications, N. Mastorakis (ed.), World Scientific and Engineering Society Press, (2000) 163-168
6. Holcombe M. and Ipate F.: Correct Systems: Building a Business Process Solution, Springer Verlag, London (1998)
7. Eleftherakis G., Kefalas P., Towards Model Checking of Finite State Machines Extended with Memory through Refinement, Advances in Signal Processing and Computer Technologies, G.Antoniou, N.Mastorakis, O.Panfilov (eds.), World Scientific and Engineering Society Press (2001) 321-326
8. Balaneascu T., Cowling A.J., Gheorgescu H., Gheorghe M., Holcombe M. and Vertan C.: Communicating Stream X-machines Systems are no more than X-machines, Journal of Universal Computer Science, Vol. 5, no. 9 (1999) 494-507
9. Gheorgescu H., and Vertan C.: A New Approach to Communicating X-machines Systems, Journal of Universal Computer Science, Vol. 6, no. 5 (2000) 490-502

10. Barnard J.: COMX: a design methodology using Communicating X-machines, *Information and Software Technology*, Vol. 40 (1998) 271-280
11. Kefalas P. and Kapeti E.: A Design Language and Tool for X-machines Specification, In *Advances in Informatics*, D.I. Fotiadis, S.D. Nikolopoulos (eds.), World Scientific Publishing Company (2000) 134-145
12. Kefalas P.: Automatic Translation from X-machines to Prolog, TR-CS01/00, Dept. of Computer Science, CITY Liberal Studies (2000)
13. Kefalas P. and Sotiriadou A.: A complier that transforms X-machines specification to Z, TR-CS06/00, Dept. of Computer Science, CITY Liberal Studies (2000)
14. Kefalas P., Eleftherakis G. and Kehris E.: Modular System Specification using Communicating X-Machines, TR-CS11/00, Dept. of Computer Science, CITY Liberal Studies (2000)
15. Georghes M., Holcombe M. and Kefalas P.: Computational Models for Collective Foraging, *BioSystems*, 61 (2001) 133-141
16. Kefalas P.: Formal Modelling of Reactive Agents as an Aggregation of Simple Behaviours, *Lecture Notes in Artificial Intelligence* 2308, I.P.Vlahavas and C.D.Spyropoulos (eds.), Springer-Verlag, (2002) 461-472