Communicating X-Machines: A Practical Approach for Formal and Modular Specification of Large Systems

Petros KEFALAS[1], George ELEFTHERAKIS[1], and Evangelos KEHRIS[2]

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

{kefalas,eleftherakis}@city.academic.gr

[2]Technological Educational Institute
of Serres
Business Adminitration Dept.
Terma Magnisias, 62124 Serres, Greece

kehris@teiser.gr

Abstract. *An X-machine is a general computational machine that can model: (a) non-trivial data structures as a typed memory tuple and (b) the dynamic part of a system by employing transitions, which are not labeled with simple inputs but with functions that operate on inputs and memory values. The X-machine formal method is valuable to software engineers since it is rather intuitive, while at the same time formal descriptions of data types and functions can be written in any known mathematical notation. These differences allow the X-machines to be more expressive and flexible than a Finite State Machine. In addition, a set of X-machines can be viewed as components, which communicate with each other in order to specify larger systems. This paper describes a methodology as well as an appropriate notation, namely XMDL, for building communicating X-machines from existing stand-alone X-machine models. The proposed methodology is accompanied by an example model of a traffic light junction, which demonstrates the use of communicating X-machines towards the incremental modeling of large-scale systems. It is suggested that through XMDL, the practical development of such complex systems can be split into two separate activities: (a) the modeling of stand-alone X-machine components and (b) the description of the communication between these components. The approach is disciplined, practical, modular and general in the sense that it subsumes the existing methods for communicating X-machines.*

1. Introduction

Although many software engineering methods and methodologies are devised in order to deal with the development of complex software systems, there is still no evidence to suggest that, apart from formal methods, any of them leads towards "correct" systems. In the last few decades, academics and practitioners adopted extreme positions for or against formal methods [1], with the truth lying somewhere between but the necessity of formal methods in software engineering of industrial systems still apparent [2]. Software system specification has centred on the use of models of data types, either functional or relational models such as Z [3] or VDM [4] or axiomatic ones such as OBJ [5]. Although these have led to some considerable advances in software design, they lack the ability to express the dynamics of the system. Also, transforming an implicit formal description into an effective working system is not straightforward. Other formal methods, such as Finite State Machines [6] or Petri Nets [7] capture the dynamics of a system, but fail to describe the system completely, since there is little or no reference at all to the internal data and how this data is affected by each operation in the state transition diagram. Other methods, like Statecharts [8], capture the requirements of both the dynamic behaviour and modelling of data but are rather informal with respect to clarity and semantics, thus being susceptible to many interpretations. So far, little attention has been paid in formal methods that could facilitate all crucial stages of "correct" system development, namely modelling, verification and testing.

X-machines is a formal method that is able to deal with all these crucial stages. An X-machine is a general computational machine introduced by Eilenberg [9] and extended by Holcombe [10], that resembles a Finite State Machine (FSM) but with two significant differences: (a) there is an underlying data set attached to the machine, and (b) the transitions are not labeled with simple inputs but with functions that operate on inputs and data set values. These differences allow the X-machines to be

more expressive and flexible than the FSM. In this paper, we use X-machines for modeling communicating systems.

As described above, the majority of formal languages facilitate the modeling of either the data processing or the control of a system. A particular interesting class of X-machines is the Stream X-machines that can model non-trivial data structures as a typed memory tuple. Stream X-machines employ a diagrammatic approach of modeling the control by extending the expressive power of the FSM. They 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, which is held in the memory. 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 [11] is an 8-tuple $M = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, where:

- Σ, Γ is the input and output finite alphabet respectively, i.e. two sets of symbols,
- Q is the finite set of states,
- M is the (possibly) infinite set called memory, i.e. an n-tuple of typed symbols,
- Φ is the type of the machine M, a finite set of partial functions φ 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 Φ, denotes the next state. F is often described as a transition state diagram, $F: Q \times \Phi \rightarrow Q$, and
- q_0 and m_0 are the initial state and memory respectively.

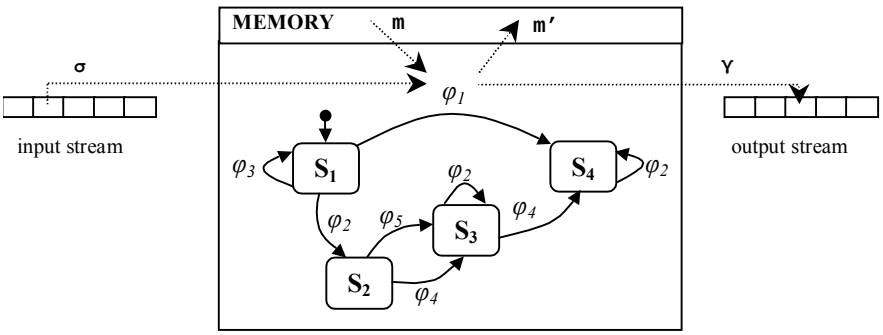


Fig. 1. An abstract example of an X-machine; φ_i: functions operating on inputs and memory, S_i: states. The general format of functions is: $\varphi(\sigma,m) = (\gamma,m')$ if condition.

Stream X-machines can be thought to apply in similar cases where Statecharts and other similar notations, such as SDL, do. However, apart from being formal as well as proved to possess the computational power of Turing machines [11], X-machines have other significant advantages. Firstly, they provide a mathematical modeling formalism for a system. Consequently, they offer a strategy to test the implementation against the model [12,13], which is a generalization of W-method for FSM testing [14]. It is proved that this testing method is guaranteed to determine correctness if certain assumptions in the implementation hold [11]. Finally, a model checking methodology for X-machines is devised [15] that facilitates the verification of safety properties of a model. Therefore, the 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 [15,16]. In principle, X-machines are considered a generalization of models written in similar formalisms. Concepts devised and findings proven for X-machines form a solid theoretical framework, which can be adapted to other, more tool-oriented methods, such as Statecharts or SDL.

In this paper, we explore another dimension of X-machine modeling, i.e. the ability to specify large-scale systems in terms of components that communicate with each other. In addition, we demonstrate how formal specification can also be practical through an appropriately devised notation and an incremental modular development methodology. In section 2 of this paper, a review of the communi-

cating X-machine approaches is presented and the motivation of our work is given by identifying the limitations of existing approaches. The main contribution of this work is analytically discussed in section 3 where our methodology as well as the appropriate notation is described. A concrete example is given in order to accompany the theory and demonstrate the applicability of the approach. In section 4, the advantages of the current approach over the alternatives are discussed. Finally, in the last section, current as well as further work concludes this paper.

2. Review of Communicating X-machine Theory

A number of different communicating X-machine approaches have been proposed [17,18,19]. We describe in more detail the approach proposed in [17], since, it is more general than [18], and unlike [19], it is sound and preserves the main advantage of X-machines, i.e. the testing method for stand-alone X-machines can be applied. All the approaches are compared to the proposed methodology of this paper in the last section. A Communicating Stream X-machine with n components is a system [17]:

$$CSXM_n = ((XMC_i)_{i=1..n}, CM, C_0)$$

where:

- XMC_i is an X-machine Component of the system,
- CM is a $n \times n$ matrix, namely the Communication Matrix,
- C_0 is the initial communication matrix.

In the following, we will refer to components of the communicating stream X-machine as XMC rather than X-machine, since the definition of XMC is different from the one of a stand-alone X-machine. In the described model, the communication of XMCs is established through the communication matrix. The matrix cells contain "messages" from one XMC to another. The (i,j) cell contains a "message" from XMC_i to XMC_j, i.e. XMC_i reads only from i^{th} column and writes only in i^{th} row. The matrix cells may contain an undefined value λ which stands for "no message" while all the (i,i) cells remain empty. The "messages" can be of any type defined in all XMC memories. In order to deliver or receive "messages", an XMC_i requires an input port IP_i and an output port OP_i, which contain individual elements (Fig. 2). The type of these elements is IN_i and OUT_i respectively, which are sets of values from the memory M_i or the undefined value λ, that is $IN_i, OUT_i \subseteq M_i \cup \{\lambda\}$ and $\lambda \notin M_i$.

In order to utilize the IP and OP ports, there exist special kind of states and functions, which are called Communicating States and Communicating Functions respectively. Therefore, inside an XMC, there exist:

- Processing Functions, which affect the contents of IP and OP ports, emerge from processing states, and do not affect the communication matrix, and
- Communicating Functions, which either read an element from the matrix and put it in the IP port, or write an element from the OP port to the matrix.

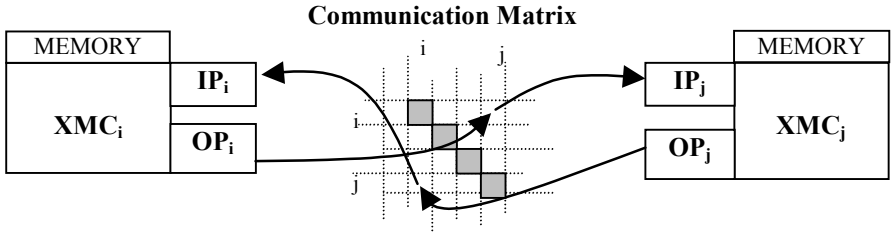


Fig. 2. Two X-machines Components, XMC_i and XMC_j can communicate through their IP and OP ports and the Communication Matrix.

The communicating functions emerge only from communication states, accept the empty symbol ε as input, and produce ε as output, while not affecting the memory. The communicating functions can write to the matrix only if the cell contains the special value λ. After the communicating functions read from the matrix, the cell is assigned the value λ. If a communication function is not applicable, it

"waits" until it becomes applicable (Fig. 3). Formally, the processing functions of an XMC are defined as:

pf(σ, m, in, out) = (γ, m', in', out') where: $\sigma \in \Sigma$, $m,m' \in M$, $in,in' \in IN$, $out,out' \in OUT$, $\gamma \in \Gamma$

and, the communication functions are defined as:

cf(ε, m, in, out, c) = (ε, m, in', out', c') where: $m \in M$, $in,in' \in IN$, $out,out' \in OUT$, $c,c' \in CM$

Therefore, the new Φ_i set for an XMC_i is the union of all the processing functions $pf \in \Phi_{pi}$ with all the communicating functions $cf \in \Phi_{ci}$, i.e. $\Phi_i = \Phi_{pi} \cup \Phi_{ci}$ and $\Phi_{pi} \cap \Phi_{ci} = \varnothing$, where:

$pf : \Sigma_i \times M_i \times IN_i \times OUT_i \rightarrow \Gamma_i \times M_i \times IN_i \times OUT_i$

$cf: \Sigma_i \times M_i \times IN_i \times OUT_i \times CM \rightarrow \Gamma_i \times M_i \times IN_i \times OUT_i \times CM$

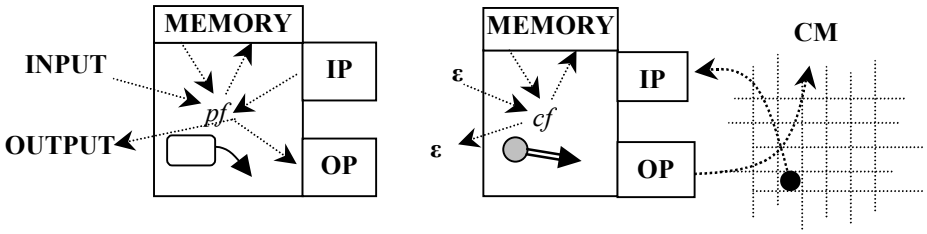


Fig.3. An abstract view of Processing and Communicating Functions and their parameters.

For example, assume a producer and a consumer combination. The producer generates items, while the consumer accepts two items and operates on them. The two XMCs, together with their processing as well as communicating states and functions are illustrated in Fig. 4. The producer is an XMC that, given a stream of inputs, activates the processing functions *begin* and *continue*, which in turn store a partially constructed item into the memory. When the item construction is *complete*, the item is transferred to the OP port and the machine is at *transmitting* state. The function *send* is activated in order to send the item from the OP port to the communication matrix. On the other hand, the consumer is an XMC that is at *waiting* state until an item appears in the communication matrix. When it does, the function *get* is activated, putting the item into the IP port. The processing function *accept first* is responsible for moving the item from the IP port to the memory. When the second item is read, *consume* operates on both items existing in the memory. A formal definition of the two XMCs exists but falls outside the scope of this section.

It is apparent that the above XMCs are different from stand-alone X-machines that someone could specify without the intention to communicate, since:

- there might be different initial states, e.g. q_0=*waiting* for $XMC_{consumer}$ while for the stand-alone X-machine would have been q_0=*consuming_first,*
- there are additional communicating functions in all XMCs,
- the processing functions are different, e.g. in the $XMC_{producer}$ *complete* is required to write to the IP port,
- the state transition diagrams are different.

The above approach to building communicating systems is sound and preserves the ability to generate a complete test set for the system, thus guarantying its correctness. However, it suffers one major drawback, that is, 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 stand-alone X-machine differs significantly from the definition of an XMC. Moreover, the semantics of the functions affecting the communication matrix impose a limited asynchronous operation of an XMC. For example, the operation of the $XMC_{producer}$ is restricted, since it is not allowed to construct a second item if the first one is not received by the $XMC_{consumer}$ (the cell in the communication matrix is not empty and therefore the producer keeps on being at state *transmitting*).

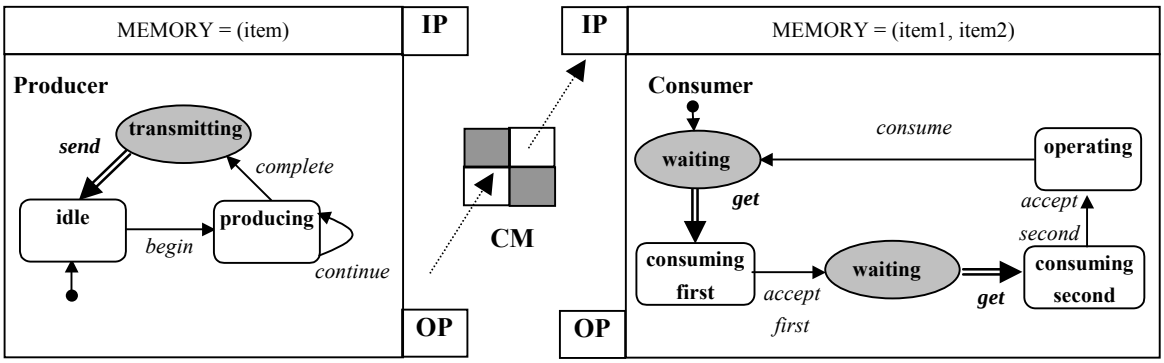


Fig.4. A producer and a consumer as XMCs of a communicating system.

A different methodology, namely COMX, of constructing communicating X-machines is described in [19]. The communication is also based in nominated IN and OUT ports and described as a separate diagram. The methodology follows a top-down approach and the intention is mainly 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 component testing as well as potential communicating system testing are unusable.

3. Building Systems from stand-alone X-machines

Our intention is to present an alternative approach for communicating X-machines and at the same time aim towards practical and disciplined development. Instead of redefining X-machines as XMCs above, we conform to the standard definition, so that X-machine specifications can be used as components of large-scale communicating systems as they are, without changes. As opposed to the previously defined approach as well as COMX, the new approach has several advantages for the developer who:

- does not need to model a communicating system from scratch,
- can re-use existing models,
- can consider modeling and communication as two separate distinct activities in the development of a communicating system,
- can use existing tools for both stand-alone and communicating X-machines.

Together with keeping the standard definition of X-machines, we suggest a bottom-up methodology for developing communicating systems, which consists of three steps:

- developing X-machine models independently of the target system, or using existing models as they are, as components of the new system,
- determining the way in which the independent models communicate,
- extending the system to accommodate more instances of already defined models.

The steps of the methodology are described in the next section through an example.

3.1 Modeling Two Independent X-machines (Step 1)

Consider a junction with three traffic lights and three corresponding queues of cars (Fig.5). The first step of the proposed methodology implies the modeling of two of system components, i.e. an X-machine for a traffic light and an X-machine for a queue of cars. The memory of a queue X-machine holds a sequence of cars arrived at the traffic light. Functions are activated by the arrival of a car or by a signal *car_leaves* when the car leaves the junction and consequently the queue. A second X-machine models a traffic light, which has two colours (red and green). The memory of the traffic light

X-machine holds the total number of ticks elapsed since the last change of colour as well as the number of ticks that a colour should be displayed (duration). An optional feature, namely the time delay, i.e. the number of ticks that should elapse before the normal operation of the traffic light may also be included in the memory. Such feature may not be specified, but, either way, it can be implemented later on in the communicating system using a different approach, as it will be shown later. The functions are activated by an input signal *tick*, which simulates a clock tick. The next state functions for the models queue and traffic light are shown diagrammatically in Fig. 6.

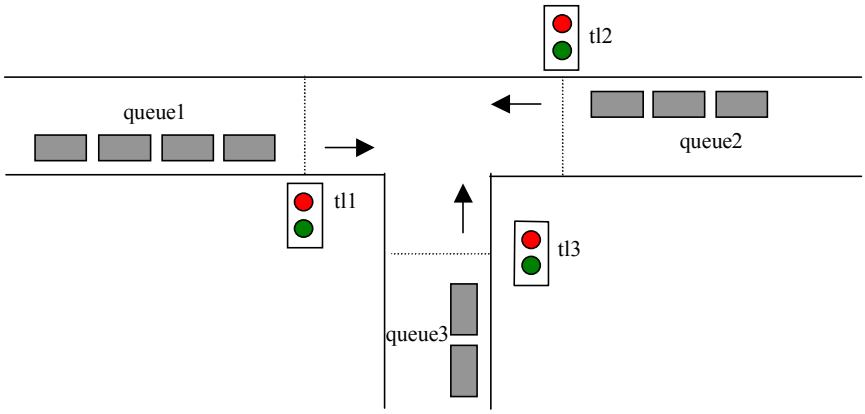


Fig.5. A junction with three traffic lights and corresponding queues.

The formal definitions of the two X-machines are presented below using the notation of X-machine Description Language [20], which is intended to be an ASCII-based interchange language. The use of XMDL makes formal specification more practical, since models that are specified in this language can be processed by various tools, such as an animator, a test set generator, a model checker etc. [21,22]. 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 informative *where* in combination with the operator <- is used to describe operations on memory values. Therefore the functions are of the form:

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

The full syntax and semantics of XMDL can be found in [23]. In the following, only the XMDL code of the queue X-machine will be analytically described, while the code of the traffic light will be just listed. Firstly, an XMDL code includes the declarations of the model name as well as the basic and user-defined types. In this case, a *CAR* is a basic type, i.e. anything, *car_queue* is defined as a sequence of cars. The types *inputset* and *messages* will be used later on.

```
#model queue.
#basic_type [CAR].
#type car_queue = sequence_of CAR.
#type inputset = CAR union {car_leaves}.
#type messages={FirstArrived,NextArrived,CarLeft,LastCarLeft,NoCarInQueue}.
```

The memory of the queue X-machine should hold the sequence of cars. The initial memory is declared as an instance of memory, in which the queue is empty.

```
#memory (car_queue).
```

```
#init_memory (nil).
```

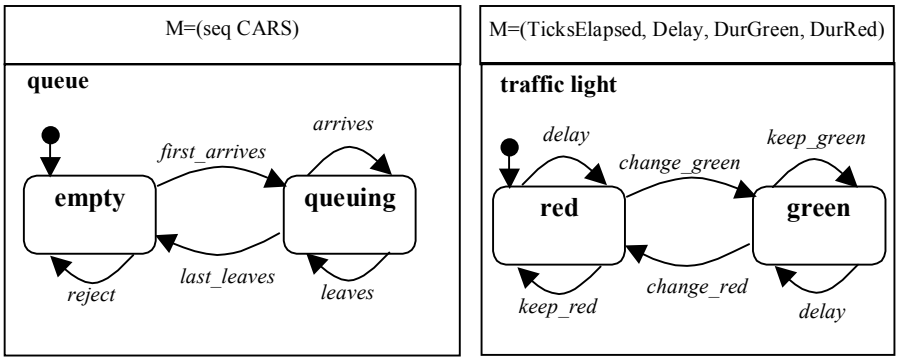


Fig.6. The state transition diagrams of the queue and traffic light X-machines

Accordingly, the set of states and the initial state are declared:

```
#states = {empty, queuing}.
#init_state {empty}.
```

The input and output symbols are based on the types previously declared by the specifier. The input is declared as any *car* or the signal *car_leaves*, whereas the output indicates the operation triggered:

```
#input (inputset).
#output (messages).
```

The set of transitions shown in Fig. 6 are listed:

```
#transition (empty, first_arrives) = queuing.
#transition (queuing, arrives) = queuing.
#transition (queuing, leaves) = queuing.
#transition (queuing, last_leaves) = empty.
#transition (empty, reject) = empty.
```

The function *first_arrives* is triggered by an input, i.e. a car, when the queue is empty. As a result a car is put in the queue sequence:

```
#fun first_arrives( (?c), (nil)) =
     if ?c belongs CAR then ((FirstArrived), (<?c>)).
```

In order to complete the specification, the rest of the functions are defined in the same manner:

```
#fun arrives( (?c), (?queue)) =
     if ?c belongs CAR then ((NextArrived),(?newqueue))
where
     ?newqueue <- ?c addatendof ?queue.
#fun leaves( (car_leaves), (<?c :: ?rest>)) = ((CarLeft), (?rest)).
#fun last_leaves( (car_leaves), (<?c>)) = ((LastCarLeft), (nil)).
#fun reject( (car_leaves), (nil)) = ((NoCarInQueue), (nil)).
```

Accordingly the traffic light X-machine is defined as follows:

```
#model traffic_light.
#type ticks_elapsed = natural0.
#type delay = natural0.
#type duration_green = natural.
#type duration_red = natural.
#type inputsignal = {tick}.
#type output = {RedColour, GreenColour, StartUp}.
```

```
#memory (ticks_elapsed, delay, duration_green, duration_red).
#states = {red, green}.
#init_memory (0,10,30,20).
#init_state = {red}.
#input (inputsignal).
#output (outputset).
#transition (red, keep_red) = red.
#transition (red, change_green) = green.
#transition (red, delay) = red.
#transition (green, keep_green) = green.
#transition (green, change_red) = red.
#transition (green, delay) = green.

#fun delay ((tick), (?te, ?delay, ?dg, ?dr)) =
     if ?d>0 then ( (StartUp), (?te, ?new_delay, ?dg, ?dr) )
where ?new_delay <- ?delay - 1.
#fun keep_red ((tick), (?te,0,?dg, ?dr)) =
     if ?new_te<?dr then ( (RedColour), (?new_te, 0, ?dg, ?dr) )
where ?new_te <- ?te + 1.
#fun keep_green ((tick), (?te,0,?dg,?dr)) =
     if ?new_te<?dg then ( (GreenColour), (?new_te, 0, ?dg, ?dr) )
where ?new_te <- ?te + 1.
#fun change_green ((tick), (?dr,0,?dg,?dr)) = ((GreenColour), (0,0,?dg,?dr)).
#fun change_red ((tick), (?dg,0,?dg,?dr)) = ((RedColour), (0,0,?dg,?dr)).
```

3.2 Building a Communicating System (Step 2)

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 X-machine depends on the number of other X-machines, from which it receives messages (Fig. 7).

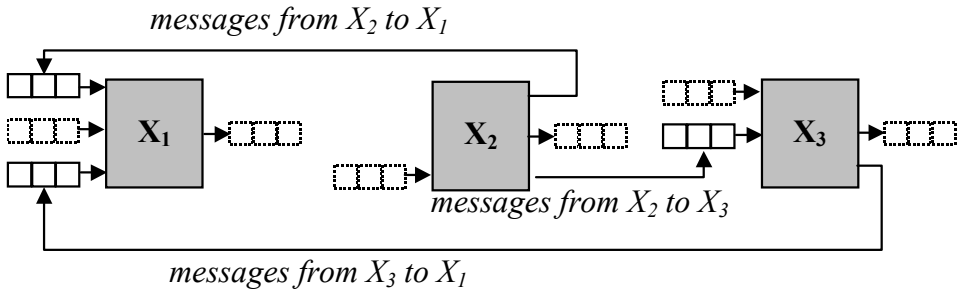


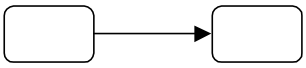
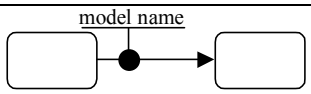
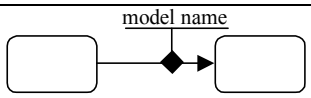
Fig. 7. Three X-machines X_1, X_2, and X_3 and the resulting communicating system where X_2 communicates (writes) with X_1 and X_3, while X_3 communicates (writes) with X_1. Therefore X_1 has three input streams, X_3 has two input streams, while X_2 has only its own standard input stream

The previously defined X-machines can communicate in the following way; when the traffic light becomes *green*, the queue can be notified to leave the cars depart one by one, providing that there is at least one car in the queue. For simplicity, we can assume that one car leaves the queue on each tick of the clock. This can be adjusted, as we shall see later. While this happens, more cars may *arrive* joining the queue, waiting for a signal to depart. The above interaction would mean that the X-machine *traffic light* should send a message to X-machine *queue*, which will act as an input.

If so annotated, the functions accept input from the communicating steam instead of the standard input stream. Also, the functions may write to a communicating input stream of another X-machine. The normal output of the functions is not affected. The annotation used to indicate communication is

depicted in Table 1. Formally speaking, the definition of type Φ in the X-machine changes from that in the definition of *M*. The formal definition of the functions in Φ of a communicating X-machine component can be found in [24]. In this paper, we aim to demonstrate the practicality of the approach.

Table 1. Annotation for diagrams of communicating X-machine specifications.

Annotation	Semantics
	Function reads an input from standard input stream and writes to output stream.
	Function reads an input from a communication input stream, i.e. a "message" sent by the annotated model name.
	Function writes a "message" to the communication input stream of machine with the annotated model name. The "message" is sent after all the output parameters are instantiated.

The models *queue* and *traffic_light* become as illustrated in Fig. 8. In order to incorporate the above semantics, the syntax of XMDL is enhanced by the following annotation:

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

The developer needs only to write is the XMDL code referring to communication, while the rest of the specification, i.e. the part referring to the X-machine components and described earlier, remains unchanged:

```
#communication of queue:
leaves reads from traffic_light.
last_leaves reads from traffic_light.
reject reads from traffic_light.

#communication of traffic_light:
change_green writes (car_leaves) to queue.
keep_green writes (car_leaves) to queue.
```

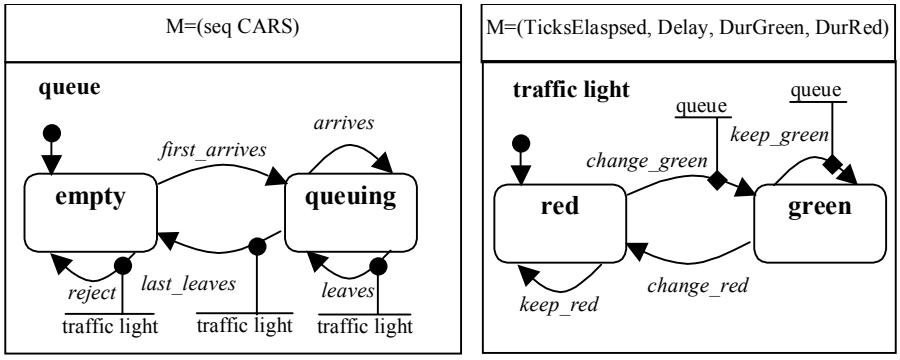


Fig.8. The state transition diagrams of the queue and traffic light X-machine specifications as a part of a communicating system

3.3 Extending the System (Step 3)

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 more traffic lights in the system, then there must be an additional X-machine that does the scheduling.

In the approaches presented by other researchers [17,18,19], the developer should start from scratch by re-defining all XMCs. In the current approach, re-building the system includes only the modification of the communication part and one new specification, i.e. the scheduler X-machine. In the following example, there are three traffic lights and therefore three queues of cars, which operated in a round robin fashion (Fig.5). The scheduler is responsible to synchronise and allocate a time-share to each device (e.g. traffic light) to operate on a certain mode (e.g. become green).

The scheduler model can either be created from scratch as an X-machine, or it may already exist as a component of some other system that needed the same type of device scheduling. The model of the scheduler is general and does not refer to any of the other X-machines, i.e. queue or traffic light. The complete model of the scheduler in XMDL is the following:

```
#model scheduler.
#type InputSet = {clock_pulse, switch_device}.
#type AccumulatedTime=natural0.
#memory (AccumulatedTime).
#states = {scheduling_device_1, scheduling_device_2, scheduling_device_3}.
#init_state = {scheduling_device_1}.
#init_memory (0).
#input (InputSet).
#output (AccumulatedTime).
#transition (scheduling_device_1, operate) = scheduling_device_1.
#transition (scheduling_device_1, switch) = scheduling_device_2.
#transition (scheduling_device_2, operate) = scheduling_device_2.
#transition (scheduling_device_2, switch) = scheduling_device_3.
#transition (scheduling_device_3, operate) = scheduling_device_3.
#transition (scheduling_device_3, switch) = scheduling_device_1.

#fun operate ((clock_pulse), (?t)) = ((?nt), (?nt))
where      ?nt <- ?t+1.
#fun switch ((switch_device), (?t)) = ((?t), (?t)).
```

The complete system is illustrated in Fig.9. The scheduler communicates with the three traffic lights by sending the "message" *tick* to them. The traffic lights communicate with the corresponding queues as shown earlier by sending the "message" *car_leaves*. Finally, a traffic light sends the "message" *switch_device* to the scheduler after the appropriate time has elapsed and changed from green to red or vice versa.

Nothing in the original models of the X-machine components that describe the queue and the traffic light need to change. However, several instances of the models *queue* and *traffic_light* should be created, each one with different initial state and initial memory. A syntax that can serve this purpose is the following:

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

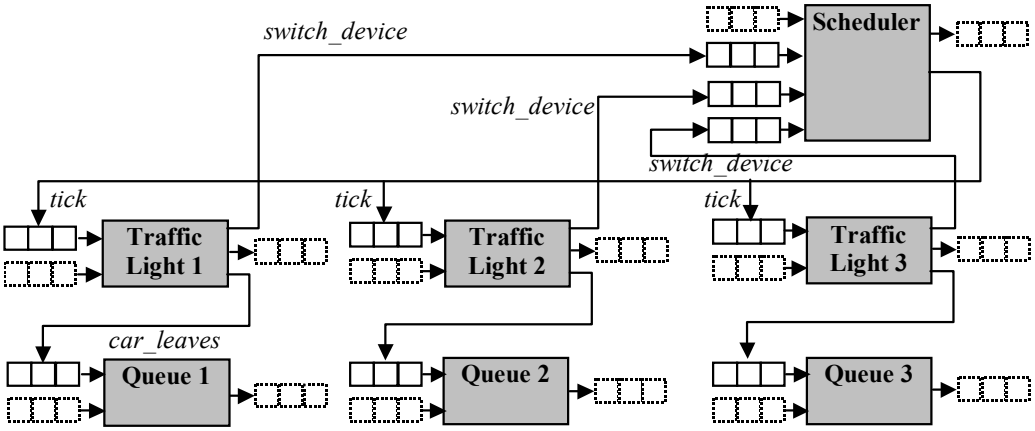


Fig. 9. The complete junction system containing three traffic lights as well as their corresponding queues and a scheduler to control them in a round robin fashion.

For the junction system, if the traffic lights have a 10-20-30 clock ticks green time-share, the XMDL code is the following:

```
#model tl1 instance_of traffic_light with:
#init_state = {green}.
#init_memory (0,0,10,50).
#model tl2 instance_of traffic_light with:
#init_state = {red}.
#init_memory (0,10,20,40).
#model tl3 instance_of traffic_light with:
#init_state = {red}.
#init_memory (0,30,30,30).
#model queue1 instance_of queue.
#model queue2 instance_of queue.
#model queue3 instance_of queue.
```

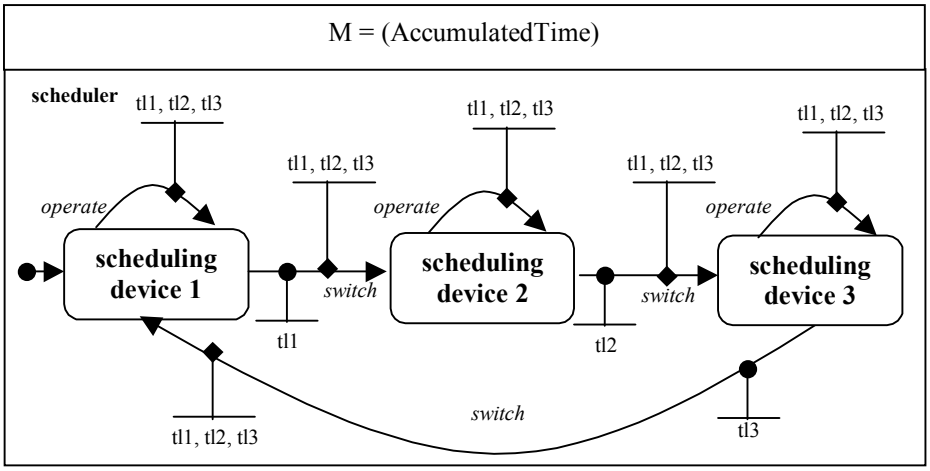


Fig.10. The state transition diagram of the scheduler X-machine specification as a part of a communicating system.

The communicating X-machine model for the scheduler is shown in Fig. 10. In addition, since the communication part is system specific, it should be written as follows:

```
#communication of scheduler:
switch reads from tl1.
switch reads from tl2.
switch reads from tl3.
switch writes (tick) to tl1.
switch writes (tick) to tl2.
switch writes (tick) to tl3.
operate writes (tick) to tl1.
operate writes (tick) to tl2.
operate writes (tick) to tl3.

#communication of tl1:
delay reads from scheduler.
keep_red reads from scheduler.
keep_green reads from scheduler.
change_red reads from scheduler.
change_green reads from scheduler.
change_red writes (switch_device) to scheduler.
change_green writes (car_leaves) to queue1.
keep_green writes (car_leaves) to queue1.

#communication of queue1:
leaves reads from tl1.
last_leaves reads from tl1.
reject reads from tl1.
```

The communication of *tl2*, *tl3* and *queue2*, *queue3* are similar to the above. The kind of interaction described above, also appears when buffering or synchronization is required. For instance, if a function of a machine M, requires an n-tuple as input, then, assuming that every element of the n-tuple is produced by a different machine M_1, …, M_n, a new buffer X-machine should be specified in order to construct the n-tuple and then pass it to machine M for consumption. In many cases, one can imagine "ready-made" generic X-machines that would act as synchronization interfaces or buffers between other machines in a communicating system. For example, if the *delay* function was not accommodated in the original model of the traffic light, then a new X-machine could be specified, which would stand between the *scheduler* and *tl2* and *tl3* and which could perform the required functionality. Another example is the assumption made above that a queue can be notified to leave the cars departing one by one at each tick of the traffic light. This can be changed by modeling a buffer that will stand between a traffic light and its corresponding queue, which will send the "message" *car_leaves* to the queue according to some other requirement.

Sumarising the proposed methodology, a communicating system can be developed in three stages: (a) modeling of X-machine components irrespectively of the target system or use off-the-shelf X-machine components, (b) modeling of the communication between these components, and (c) creation of instances of the developed components and possibly add off-the-shelf generic interfaces between components of the final system. In this incremental approach, the design of XMDL notation and its accompanying tools play an important role. One can: (a) use XMDL to model X-machine components or use existing XMDL models, (b) write a separate XMDL file which deals with the communication, and (c) use XMDL declarations for creating instances of the models. The final system is a result of the compilation of all these separate XMDL files. The compiled code represents the communicating X-machine system (Fig.11).

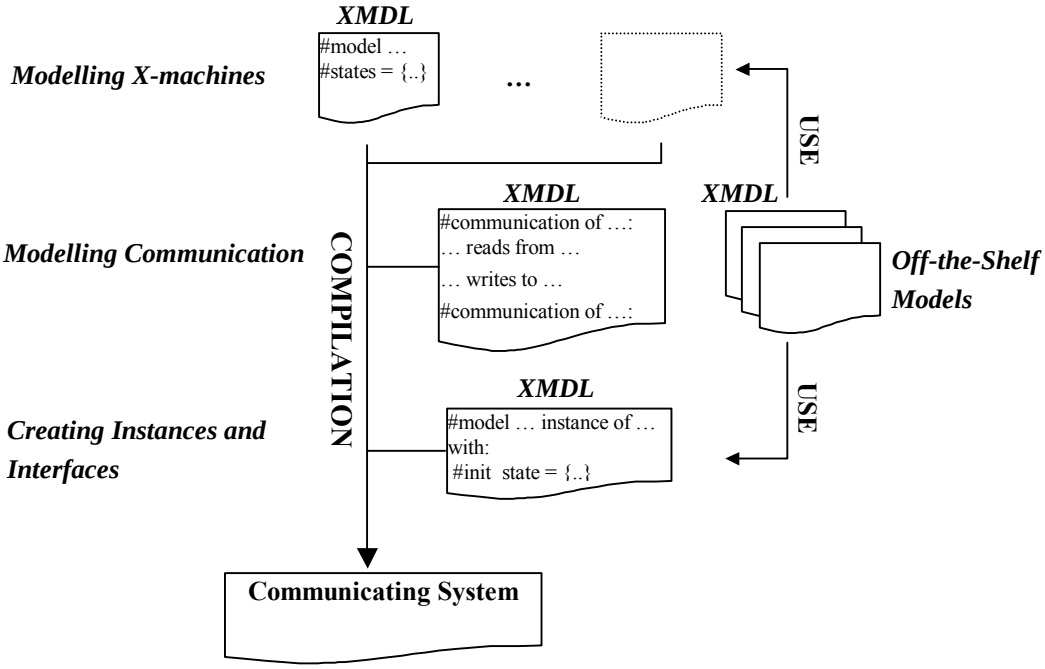


Fig.11. The stages of development of a communicating system through XMDL.

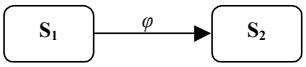
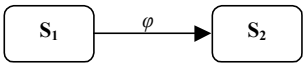
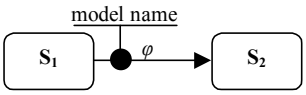
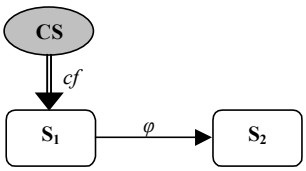
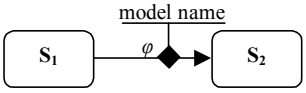
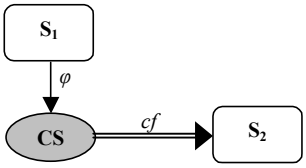
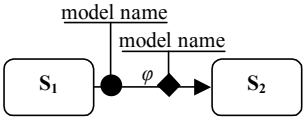
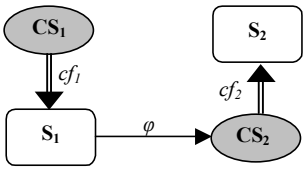
4. Relation to Other Communicating X-machine approaches

The proposed Communicating X-machine methodology is a different approach towards communicating systems. In comparison with the already suggested methodology elsewhere [17,18,19], the current approach is:

- **Practical**: in the sense that existing tools may be used based on the same notation, i.e. XMDL, for stand-alone as well as communicating systems.
- **Modular**: since 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.
- **Disciplined**: since modeling and communication are regarded as two separate developing activities.
- **General**: in the sense that it subsumes the existing approaches.
- **Asynchronous**: since the 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.

The way for describing the communication with the annotations presented earlier 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 (Table 2). This subsumption guarantees that the properties proved in [17] are also valid for the current approach.

Table.2. Transformation between the suggested annotations into X-machines with processing as well as communicating functions and states

Annotation	Explanation	Equivalent Transitions
	Processing Functions remain the same in both methods.	
	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 IP port. After that the processing function takes an empty input.	
	Function that writes a message to another machine's communication input stream, can be viewed as a processing state that writes to the OP port, followed by a communicating state which in turn is followed by a communication function that writes to the matrix.	
	From the above two the equivalent is implied.	

5. Conclusions

We have presented a methodology for building communicating systems out of existing components, i.e. stand-alone X-machines. The approach is practical, in the sense that 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 systems. Also, X-machine models can be re-used in other systems, since the only part that needs to be changed is the communication. 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 can be tested and verified separately [11,15]. Also, the current approach is more general than the more recent communicating model which aims towards a testing strategy for a communicating system [25]. It is therefore feasible to translate the communicating system into a form that complete testing can be applied, as explained in [25].

We have applied the methodology for developing formal models for simulation [13,24,26]. It is found that by using communicating X-machines, we can formally model multi-agent systems, with agents modeled as an aggregate of different communicating behaviours [27, 28]. Future work will include the implementation of communicating systems on top of the already existing tools. In addition, a theoretical framework is under development in order to prove the subsumption of all other models proposed for Communicating X-machines by the current approach. Finally, a methodology for model checking of large scale communicating systems and their safety properties, as in stand-alone X-machines will be investigated.

References

1. Young W. D., Formal Methods versus Software Engineering: Is There a Conflict? In Proceedings of the Fourth Testing, Analysis, and Verification Symposium, (1991) 188-899
2. Clarke E. and Wing J. M., Formal Methods: State of the Art and Future Directions, ACM Computing Surveys, Vol.28, No.4, (1996) 626-643
3. Spivey M., The Z notation: A reference manual. Prentice-Hall, 1989
4. Jones C. B., Systematic software development using VDM (2nd ed.). Prentice-Hall, 1990
5. Futatsugi K., Goguen, J., Jouannaud, J.-P., and Meseguer, J., Principles of OBJ2. In B.Reid (Ed.). Proceedings, 12th ACM Symposium on Principles of Programming Languages, (1985) 52-66
6. Wulf W. A., Shaw M., Hilfinger P.N., and Flon L., Fundamental structures of computer science. Addison-Wesley, 1981
7. Reisig W., Petri nets - an introduction. EATCS Monographs on Theoretical Computer Science, 4, Springer-Verlag, 1985
8. Harel D., Statecharts: A visual approach to complex systems. Science of Computer Programming, Vol.8, No.3, (1987) 231-274
9. Eilenberg S., Automata Machines and Languages, Vol. A, Academic Press, 1974.
10. Holcombe M., X-machines as a basis for dynamic system specification, Software Engineering Journal, Vol.3, No.2 (1988) 69-76
11. Holcombe M. and Ipate F., Correct Systems: Building a Business Process Solution, Springer Verlag, London (1998)
12. 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
13. 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
14. Chow T.S., Testing Software Design Modeled by Finite-State Machines, IEEE Transactions on Software Engineering, Vol.SE-4, No.3 (1978) 178-187
15. Eleftherakis G., Kefalas P. and Sotiriadou A., XmCTL: Extending Temporal Logic to Facilitate Formal Verification of X-Machines, Matematica-Informatica, Analele Universitatii Bucharest, Vol.50, (2002) 79-95
16. Eleftherakis G. and 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
17. 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
18. Cowling A.J., Gheorgescu H., and Vertan C., A Structured Way to use Channels for Communication in X-machines Systems, formal Aspects of Computing, Vol. 12, (2000) 485-500
19. Barnard J., COMX: a design methodology using Communicating X-machines, Journal of Information and Software Technology, Vol. 40 (1998) 271-280
20. 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
21. Kefalas P., Automatic Translation from X-machines to Prolog, Technical Report TR-CS01/00, Dept. of Computer Science, CITY Liberal Studies (2000)
22. Kefalas P. and Sotiriadou A., A Compiler that transforms X-machines specification to Z, Technical Report TR-CS06/00, Dept. of Computer Science, CITY Liberal Studies (2000)
23. Kefalas P., XMDL User Manual, Version 1.6, Technical Report TR-CS07/00, Dept. of Computer Science, CITY Liberal Studies (2000)

24. Kefalas P., Eleftherakis G., Holcombe M. and Gheoghe M., Simulation and Verification of P Systems through Communicating X-machines, BioSystems (to be published)
25. Ipate F., Balanescu T., Kefalas P., Holcombe M. and, Eleftherakis G., A New Model for Communicating X-machines, Romanian Journal of Information Technology (to be published)
26. Kefalas P., Eleftherakis G. and Kehris E., Modular Modeling of Large-Scale Systems using Communicating X-Machines, Proceedings of the 8th Panhellenic Conference in Informatics, Cyprus, (2001)
27. Georghe M., Holcombe M. and Kefalas P., Computational Models for Collective Foraging, BioSystems 61, (2001) 133-141
28. 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.