

Modular Modeling of Large-Scale Systems using Communicating X-Machines

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. A X-machine is a general computational machine that can model non-trivial data structures as a typed memory tuple as well as the dynamic part of a system by employing transitions labeled with functions that operate on inputs and memory values. The X-machine formal method is rather intuitive, while at the same time formal descriptions of data types and functions can be written in any known mathematical notation. A set of X-machines can be viewed as components, which communicate with each other. This paper describes a methodology of building communicating X-machines from existing stand-alone X-machine models. It is suggested that the development of 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 methodologies. 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

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. In this paper, we use X-machines for modeling communicating systems.

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:

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

- Σ, Γ is the input and output finite alphabet respectively,
- Q is the finite set of states,
- M is the (possibly) infinite set called memory,
- 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.
- $F: Q \times \Phi \rightarrow Q$
- q_0 and m_0 are the initial state and memory respectively.

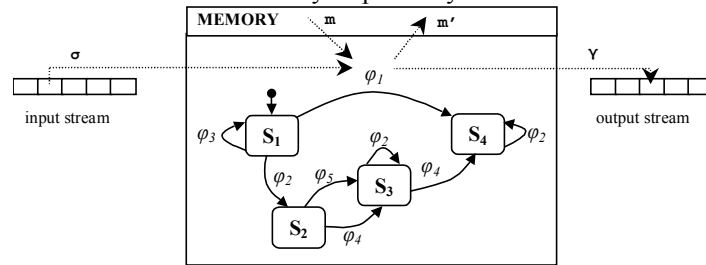


Fig. 1. An abstract example of a X-machine; φ_i : functions operating on inputs and memory, S_i : states.

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 to 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].

In section 2, a review of the communicating X-machine approaches is presented and the motivation of our work is given. Our main contribution is analytically discussed in section 3 where our methodology is described. A concrete example is given in order to demonstrate the applicability of the approach. In section 4, the advantages of the current approach over the alternatives are discussed. Finally, ideas for further work conclude this paper.

2 Communicating X-machines Theory

A number of communicating X-machine approaches have been proposed [8,9,10]. A n component communicating stream X-machine is a tuple $CSXM_n = ((XMC_i)_{i=1..n}, CM, C_0)$ where [8]:

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

A component XMC_i of $CSXM_n$ is different from a stand-alone X-machine, since they utilize an IN and an OUT port for communication (Fig.2). Both ports are linked to CM , which acts as the communication means between $XMCs$. The CM cells contain “messages”, i.e. the cell (i,j) contains a “message” from XMC_i to XMC_j . The special value λ stands for “no message”. The “messages” can be of any type defined in all XMC memories. In addition, there exist special kind of states and functions. 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 either read an element from CM and put it in the IN port, or write an element from the OUT port to the CM :

$cf(\varepsilon, m, in, out, c) = (\varepsilon, m, in', out', c')$ where: $m, m' \in M, in, in' \in IN, out, out' \in OUT, c, c' \in CM$
 The communicating functions can write to the matrix only if the cell contains the special value λ . After the communicating functions read from CM , the cell is assigned the value λ . 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:

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

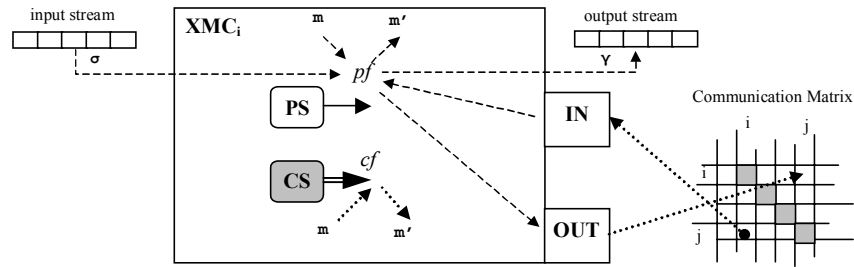


Fig. 2. An abstract example of a XMC_i with IN and OUT ports and the Communication Matrix

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.

The above approaches to 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 differs significantly from the definition of a XMC. Moreover, the semantics of the functions affecting the communication matrix impose a limited asynchronous operation of a XMC.

3 Building Systems from stand-alone X-machines

The alternative approach is to use stand-alone X-machine models as components of a large-scale communicating system. The approach consists of two steps: (a) develop X-machine models independently of the target system, or use existing models as they are, as components, and (b) determine the way in which the independent models communicate.

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.

3.1 Modeling Two Independent X-machine (Step 1)

Assume a X-machine, which models a queue of customers (Fig.3). The memory holds a queue of customers and the total number of customers arrived. Functions are activated by the arrival of a customer or by a signal *customer_leaves* when the customer leaves the queue. A second X-machine models a cashier who serves persons (Fig.3). The memory holds the total number of persons served. The functions are activated by input signals *start_service* and *finish_service* together with the person who is currently being served. The complete X-machine models can be found in [14].

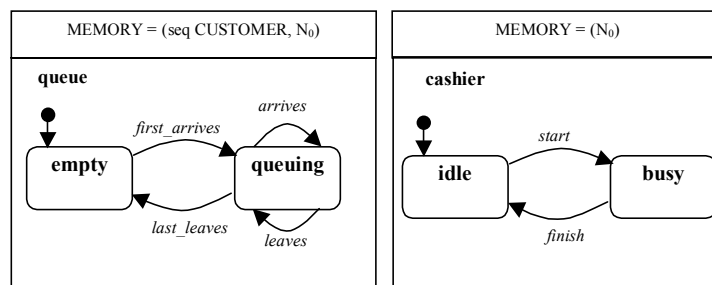


Fig.3. The state transition diagrams of the queue and cashier X-machines

The formal definitions of the two X-machines are presented below 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 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>.
```

Firstly, a XMDL code includes the declarations of the model name as well as the basic and user-defined types. In this case, a *CUSTOMER* is a basic type, i.e. anything, *customer queue* is defined as a sequence of customers, and *customers_arrived* is a counter for keeping track of the total number of customers joined the queue.

```
#model queue.
#basic_type [CUSTOMER].
#type customer_queue = sequence_of CUSTOMER.
#type customers_arrived = natural0.
#type messages={FirstArrived,NextArrived,CustLeft,LastLeft}.
```

The memory of the queue X-machine should hold the sequence of customers and the total number of customers. The initial memory is declared as an instance of memory, in which the queue is empty and there is no customer ever joined the queue. Accordingly, the set of states and the initial state are declared.

```
#memory (customer_queue, customers_arrived).
#init_memory (nil, 0).
#states = {empty, queuing}.
#init_state {empty}.
```

The input is any *customer* or the signal *customer_leaves*, whereas the output is a tuple indicating the operation triggered and the total number of customers joined the queue:

```
#input (CUSTOMER union {customer_leaves}).
#output (messages, customers_arrived).
```

The set of transitions shown in Fig. 3 are listed in the form:

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

The function *first_arrives* is triggered by an input, i.e. a customer, when the queue is empty. As a result a customer is put in the queue and the number is increased by one:

```
#fun first_arrives( (?c), (nil, ?ca)) =
  if ?c belongs CUSTOMER then ((FirstArrived,?newca), (<?c> ,?newca))
  where      ?new_ca <- ?ca+1.
```

The rest of the functions are defined in the same manner accordingly:

```

#fun leaves( (customer_leaves), ( <?c :: ?rest>, ?ca)) =
    ((CustLeft, ?ca), ( ?rest, ?ca )).
#fun last_leaves( (customer_leaves), ( <?c>, ?ca)) =
    ((LastLeft, ?ca), ( nil, ?ca )).
#fun arrives( (?c), (?queue, ?ca)) =
    if ?c belongs CUSTOMER then
        ((NextArrived, ?newca), (?newqueue, ?newca ))
where
    ?newqueue <- ?c addatendof queue and
    ?new_ca <- ?ca+1.

```

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. 4).

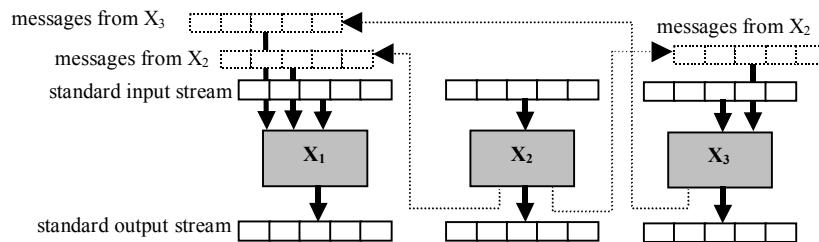


Fig. 4. Three X-machines X_1 , X_2 , and X_3 and the resulting communicating system where X_2 communicates (writes) with X_1 and X_3 , while X_3 communicates (writes) with X_1 . Therefore X_1 has three input streams, X_3 has two input streams, while X_2 has only its own input stream

The previously defined X-machines can communicate in the following way; when the cashier is in state *idle*, it can be notified to *start* serving a customer, providing that there is at least one customer in the queue. While the cashier is *busy*, more customers may *arrive*. When the cashier *finishes* the service then the customer *leaves* the queue. The above interaction would mean that the X-machine queue should send a message to X-machine cashier, which will act as input and vice versa.

If a solid circle appears on a transition function, this function accepts input from the communicating stream 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. The normal output of the functions is not affected. The models queue and cashier become as illustrated in Fig. 5. 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>.

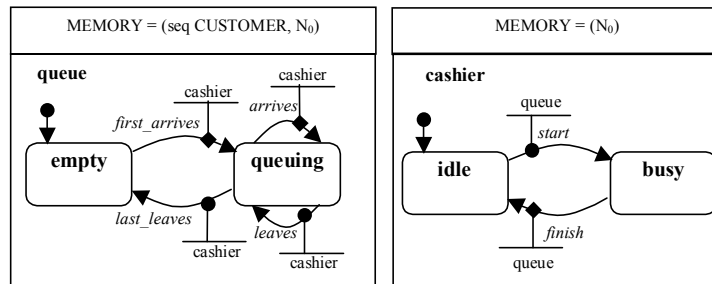


Fig.5. The state diagrams of queue and cashier specifications as a part of a communicating system

The developer needs only to write the XMDL code referring to communication:

```
#communication of queue:
last_leaves reads from cashier.
leaves reads from cashier.
first_arrives writes (start_service, ?p) to cashier
  where ? p <- head ?queue from memory (?queue, ?ca).
arrives writes (start_service, ?p) to cashier
  where ?p <- head ?queue from memory (?queue, ?ca).
#communication of cashier:
start reads from queue.
finish writes (customer_leaves) to queue.
```

3.3 Extending the 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 instances of cashiers in the system, namely cashier1 and cashier2, then there must be an additional X-machine that does the scheduling. In the approaches presented by other researchers [8,9,10], this is not easily done. In the current approach, rebuilding the system includes only the modification of the communication part and the new specification of the scheduler. The scheduler allocates a customer in the queue to a specific cashier depending on whether a cashier is idle or not (Fig. 6). 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:

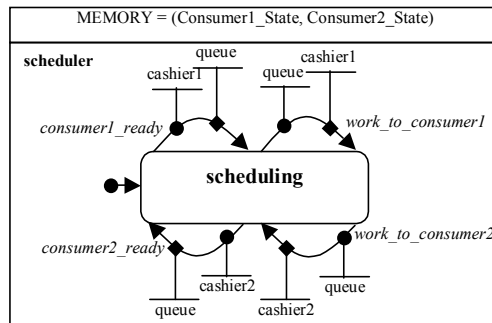


Fig.6. The transition diagram of the scheduler specification as a part of a communicating system

#communication of scheduler:

```

consumer1_ready reads from cashier1.
consumer2_ready reads from cashier2.
consumer1_ready writes (customer_leaves) to queue.
consumer2_ready writes (customer_leaves) to queue.
work_to_consumer1 reads from queue.
work_to_consumer2 reads from queue.
work_to_consumer1 writes (start_service, ?item) to cashier1
  where ?item from input (?producer, ?item).
work_to_consumer2 writes (start_servive, ?item) to cashier2
  where ?p from input (?producer, ?item).

```

The communication parts of the other three X-machines also need to change in order to accommodate the specification of the new system. For example, the queue notifies the scheduler for the arrival of the customer, while the scheduler, having recorded the state of the cashiers in its memory, assigns the customer to the one of the cashiers:

#communication of queue:

```

last_leaves reads from scheduler.
leaves reads from scheduler.
first_arrives writes (producer, ?p) to scheduler
  where ?p <- head ?queue from memory (?queue, ?ca).
arrives writes (producer, ?p) to scheduler
  where ?p <- head ?queue from memory(?queue, ?ca).

```

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.

4 Relation to Other Communicating X-machine approaches

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.
- *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 (Fig. 7). This subsumption guarantees that the properties proved in [8] are also valid for the current approach.

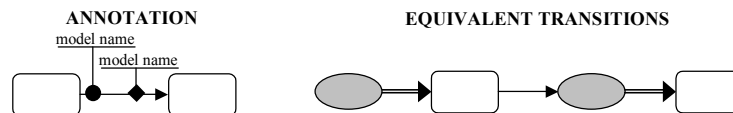


Fig.7. Equivalence of the suggested annotations to the theoretical approach

5 Conclusions

We have presented a methodology for building communicating systems out of existing stand-alone X-machines. The approach is practical, since the software engineer can separately specify the components and then describe the way in which they communicate. This allows a disciplined development of large 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 strategies in which X-machines are individually tested as components while communication is tested separately.

We are currently applying the methodology for developing formal models of multi-agent systems [15]. 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 a theoretical framework to prove the subsumption by the current approach of all other communicating X-machines approaches. 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. Georghe M., Holcombe M. and Kefalas P.: Computational Models for Collective Foraging, 4th International Workshop on Information Processing in Cells and Tissues, Lueven, Belgium (2001)