Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 561–568

ISBN 978-83-60810-22-4

A methodology for developing component-based agent systems focusing on component quality

George Eleftherakis
Petros Kefalas
Computer Science Department
CITY College, Thessaloniki, Greece
International Faculty of the University of Sheffield
Email: {eleftherakis, kefalas}@city.academic.gr

Evangelos Kehris
Department of Business Administration,
Technological Education Institute of Serres,
Greece
Email: kehris@teiser.gr

***Abstract*—Formal development of agent systems with inherent high complexity is not a trivial task, especially if a formal method used is not accompanied by an appropriate methodology. X-machines is a formal method that resembles Finite State Machines but has two important extensions, namely internal memory structure and functions. In this paper, we present a disciplined methodology for developing agent systems using communicating X-machine agents and we demonstrate its applicability through an example. In practice, the development of a communicating system model can be based on a number of well-defined distinct steps, i.e. development of types of X-machine models, agents as instances of those types, communication between agents, and testing as well as model checking each of these agents individually. To each of the steps a set of appropriate tools is employed. Therefore the proposed methodology utilises a priori techniques to avoid any flaws in the early stages of the development together with a posteriori techniques to discover any undiscovered flaws in later stages. This way it makes the best use of the development effort to achieve highest confidence in the quality of the developed agents. We use this methodology for modelling naturally distributed systems, such as multi-agent systems. We use a generalized example in order to demonstrate the methodology and explain in detail how each activity is carried out. We briefly present the theory behind communicating X-machine agents and then we describe in detail the practical issues related using the same example throughout.**

I. Introduction

AGENT oriented software engineering aims to manage the inherent complexity of distributed systems [1]. The developing process should be accompanied by methodologies and tools that can lead towards the implementation of "correct" systems: system models that match the requirements and satisfy any necessary properties in order to meet the design objectives, and system implementation that passes all tests constructed using a complete functional test generation method. All the above criteria are closely related to three stages of system development, namely modelling, verification and testing. It is argued that the use of formal methods can achieve this goal to some extend [2].

Formal modelling has centred on the use of models of data types, either functional or relational. Although these have led to some considerable advances in software design, they lack the ability to express the dynamics of the system. Some other methods, such as Finite State Machines (FSM) or Petri Nets capture the dynamics, 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 system operations. Finally, methods like Statecharts, capture the requirements of dynamic behaviour and modelling of data but are rather informal with respect to clarity and semantics. So far, little attention has been paid in formal methods that could facilitate all crucial stages of "correct" system development, modelling, verification and testing.

In this paper we use a formal method, namely X-machines and its extension Communicating X-machines, which closely suits the needs of agent-based development [3], while at the same time being practical. We present a disciplined methodology for the incremental development of simple reactive agent-based systems and we present in a formal way all the required extensions of the model which will optimize towards agent systems. The proposed methodology utilises a priori techniques (formal modelling and verification) to avoid any flaws in the early stages of the development together with a posteriori techniques (a black box formal testing strategy) to discover any undiscovered flaws in later stages. This way it makes the best use of the development effort to achieve highest confidence in the quality of the developed agents, allowing safer composition of trusted, reusable agents. The methodology is achieving all these using communicating X-machine agents as building blocks. X-machines is a formal method that enhances the class of FSM with two important characteristics, namely memory and functions. X-machine model types are defined by an input stream, an output stream, a set of values that describe their memory structure, a set of states, a state transition set and a set of functions. Labels in transitions are functions which are triggered through an input symbol and a memory instance and produce an output symbol and a new memory instance (Figure 1).

X-machines can be thought to apply in similar cases where Statecharts and other similar notations, such as SDL, do. However, X-machines have other significant advantages. Firstly, they provide a mathematical modelling formalism for a system. Consequently, a model checking method for X-machines is devised [4] that facilitates the verification of safety properties of a model. Finally, they offer a strategy to test the

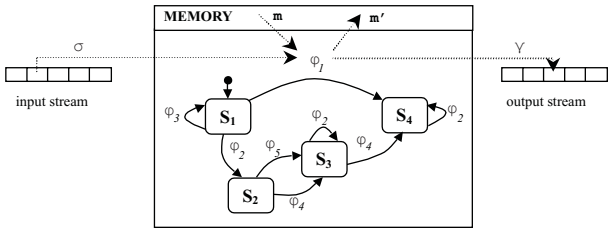

Fig. 1. An abstract X-machine model

implementation against the model [5], [6], which is a generalization of W-method for FSM testing. It is proved that this testing method is guaranteed to determine correctness if certain assumptions in the implementation hold [7]. In principle, X-machines are considered a generalization of models written in similar formalisms since 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 addition, communicating X-machines provide notation to create agents as instances of X-machine types and define their interaction and communication [8]. Functions can send messages to input streams of other X-machine agents which are consumed by local functions. In practice, it is found that the development of a communicating system model can be based on a number of well-defined distinct steps, i.e. development of types of X-machine models, creation of agents as instances of those types, construction of communicating agents, and then testing as well as model checking each of these agents individually. To each of the steps a set of appropriate tools, such as an interchange description language, parser, animator, test set generator etc., is employed in order to make the methodology applicable in real cases [9]. Such cases emerge during modelling naturally distributed systems, such as multi-agent systems. Here, we use a generalized example of a reactive agent in order to demonstrate the methodology and explain in detail how each activity, namely modelling, testing and verification is carried out. In the following, we present the methodology and in each of the sections of the paper we briefly present the theory behind each step. We then demonstrate in detail each proposed activity of the approach using a generalized example as a vehicle of study. Finally, we comment on the methodology and discuss further work to be carried out in order to deal with dynamically configurable systems as well as testing and verification of these systems as a whole.

II. METHODOLOGY

Communicating X-machines is viewed as a modelling method, where a complex system can be decomposed in small agents (elements) modelled as simple X-machine models. The communication of all these agents is specified separately in order to form the complete system as a communicating X-machine model. This implies a modular bottom-up approach and supports an iterative gradual development. It also facilitates the reusability of existing X-machine type models, making the management of the whole project more flexible and efficient, achieving its completion with lower cost and less development time.

A. Steps

The communicating X-machine method supports a disciplined modular development, allowing the developers to decompose the system under development into communicating agents and thus model interacting agent-based systems. We suggest that the development of a system model can be mapped into the following well-defined distinct actions:

(a) Develop X-machine type models (X-machine agent types) independently of the target system, or use existing type models as they are.
(b) Code the X-machine type model into a language that facilitates the subsequent steps. Use the animator that accompanies the language and get early feedback from the domain experts (informal verification).
(c) Express the desired properties in a suitable formalism (temporal logic) and use the formal verification technique (model checking) for X-machine type models in order to increase the confidence that the proposed model has the desired characteristics.
(d) Use testing strategies in order to test the implementation (Unit testing, where the unit is considered to be the agent type) against the model.
(e) Create X-machine agents and determine the way in which they communicate and interact.
(f) Extend the communicating system in order to achieve the desired overall functionality.

A set of appropriate tools has been developed and can be employed to each of the steps of the above methodology in order to make it applicable in real cases. Thus, apart from the mathematical notation used in step (a), all others are supported as follows:

- Step (b): coding of X-machine type model is carried out using the XMDL notation which acts as an interchange language for describing X-machine type models and its corresponding tools (syntax and type checker, compiler, animator)[9]. Through the animation, it is possible for the developers to informally verify that the model corresponds to the actual agent under development, and then also to demonstrate the model to the domain experts prompting them to identify any misconceptions regarding the user requirements between them and the development team.
- Step (c): formal verification of X-machine models is achieved with the use of an automated tool, a model checker. Model checking of X-machine models is supported by $\mathcal{X}m$CTL . This technique enables the designer to verify the developed model against temporal logic $\mathcal{X}m$CTL formulas which express the properties that the system should have.
- Step (d): test-cases for testing the implementation are automatically derived using the X-machine test case

generator. It is possible to use the formal testing strategy to test the implementation and prove its correctness with respect to the X-machine model.
- Step (e): the creation, communication and interaction of the agents are established through the XMDL-c notation and its corresponding tools. Achieve informal validation by demonstrating an analysis of the results from the animator for XMDL-c (simulation study) to domain experts.
- Step (f): all the above mentioned tools may be used to refine the resulting model.

There are many cases of naturally distributed multi-agent systems in which we have applied the above methodology [10], [11]. Here, we devised a generalized example of a reactive agent in order to demonstrate the methodology and explain in detail how each activity is carried out.

B. Reactive Agent Case

Reactive agents are simple agents that their responses are very closely tied to perception and they do not possess any (or they have limited) knowledge about the environment. Their behaviour can be modelled with state machines and that is why the X-machine is a perfect candidate since it can provide very elegant models of such agents [10]. For reasons of demonstration our example is an abstract and generalized one. Assume a simple reactive agent (e.g. a reactive robot, software agent) (figure 2) which consumes items (e.g. objects of a physical or artificial environment, inputs, perceptions) of the environment, processes them, and produces new items (new physical objects, output, actions/effects). Each item is uniquely identified by an identification number and its description. The simplified agent system contains two buffers which are storage spaces of limited capacity and a single agent which carries out the processing. In order to control the two buffers, two control agents are handling the communication with the processing reactive agent and the buffers. The whole agent system now can be viewed as a simple reactive agent that could communicate with other similar agent systems, forming a more complex system, providing a simple way to scale up.

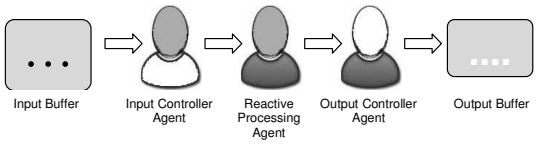

Fig. 2. Physical layout of the agent system

The items that are required to be processed by the processing agent are placed in the input buffer, while the items that have been processed by the reactive agent are stored in the output buffer. Items may be added in any buffer only if there is available space in it, while items are removed from a buffer in a specified order (e.g FIFO, LIFO or other discipline, handled by the input, output controlling agents). When the reactive agent is idle and there are items stored in the input buffer, the reactive agent may start the processing of an item: The first item p placed in the input buffer is removed from the input buffer and the reactive agent starts processing it. The processing of the item lasts for t time units. If at the completion of the item processing, the output buffer is not full, then item p is placed in the output buffer and the machines either becomes idle or starts processing another item depending on whether the input buffer is empty or not. If, however, the output buffer is full when the reactive agent completes the processing of an item, the item p may not be removed from the reactive agent and thus the reactive agent is blocked. The reactive agent is unblocked when space becomes available in the output buffer.

III. Formal Modelling X-machine Type models

A. Theory of X-machines

A deterministic stream X-machine [6] is an 8-tuple

$$X = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$$

where:
- Σ and Γ are the input and output alphabets respectively.
- Q is the finite set of states.
- M is the (possibly) infinite set called memory.
- Φ, the *type* of the machine X, is a set of partial functions φ that map an input and a memory state to an output and a possibly different memory state, $\varphi : \Sigma \times M \rightarrow \Gamma \times M$.
- F is the next state partial function, $F : Q \times \Phi \rightarrow Q$, which given a state and a function from the type Φ determines the next state. F is often described as a state transition diagram.
- q_0 and m_0 are the initial state and initial memory respectively.

The state diagram of an abstract X-machine model is shown in figure 1. An X-machine type is defined as a deterministic X-Machine without the initial state and the initial memory. Types will be used in order to create X-machine agents as shown later.

B. Mathematical Modelling

Two X-machine types are naturally identified in the manufacturing facility example, i.e. the buffer and the processing reactive agent. For example, the state transition diagrams of these two X-machine types are depicted in figure 3.

Using mathematical notation, the definition of the buffer type is as follows:

- The set of inputs is Σ =ITEMS where ITEMS = ITEM_TYPE $\times$ ID, ITEM_TYPE = { TypeA, TypeB, ... } and the set of outputs Γ =ITEMS $\times$ MESSAGES, where MESSAGES={item_removed_empty, item_removed, item_ignored,...}.
- The set of states is Q ={ empty, non_empty, full}.
- The memory is $M = \mathcal{P}$ITEMS $\times N$, with N denoting the capacity of the buffer.
- The type of the X-machine is Φ = { add_item, remove_item, become_empty, become_full, ignore_add}.

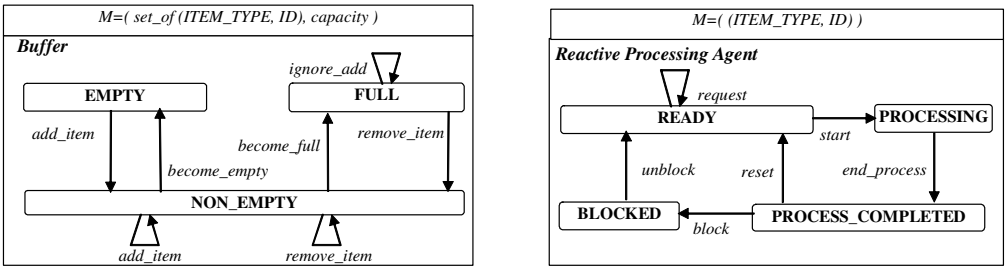

Fig. 3. X-machine model types of a processing reactive agent and a buffer

Finally, the functions $\varphi \in \Phi$ of the X-machine need to be defined. For example the function `add_item` is defined as:

$$\texttt{add\_item}\ ((type, id), (items, c)) =$$
$$(item\_added, (items \cup \{(type, id)\}, c))$$
$$if\ (type, id) \notin items \wedge card(items) + 1 \leq capacity$$

IV. Coding the Type Models

A. Modelling with XMDL

X-machine modelling is based on a mathematical notation, which, however, implies a certain degree of freedom, especially as far as the definitions of functions are concerned. In order to make the approach practical and suitable for the development of tools around X-machines, a standard notation is devised and its semantics fully defined [9]. The aim is to use this notation, namely X-machine Definition Language (XMDL), as an interchange language between developers who could share models written in XMDL for different purposes. To avoid complex mathematical notation, the language symbols are completely defined in ASCII. Briefly, an XMDL model is a list of definitions corresponding to the construct tuple of the X-machine definition. The language also provides syntax for (a) use of built-in types such as integers, sets, sequences, bags, etc., (b) use of operations on these types, such as arithmetic, logic, set operations etc., (c) definition of new types, and (d) definition of functions and the conditions under which they are applicable. In XMDL, 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 a preceding `?`. The informative `where` in combination with the operator `<-` is used to describe operations on memory values. A function has the following general syntax:

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

The following is a part of the buffer model as coded in XMDL:

```
#model buffer.
#basic_types = [ITEM_TYPE].
#type ID = Natural.
#type capacity = Natural.
#type ITEM = (ITEM_TYPE, ID).
#type buffer = set_of ITEM.

#memory (buffer, capacity).

#input (ITEM).
#output (messages, ITEM).

#type messages = {item_added,item_removed,item_ignored,
                  item_added_full,item_removed_empty}.

#states = {empty, non_empty, full}.

#transition (empty,add_item)=non_empty.
#transition (non_empty,add_item)=non_empty.
...
#fun add_item ( ((?ITEM_TYPE,?ID)), (?items, ?c) )  =
if (?ITEM_TYPE,?ID) not_belongs ?items and ?new_length < ?c
then ( (item_added, (?ITEM_TYPE,?ID)), (?new_items, ?c))
where
?new_items <- (?ITEM_TYPE,?ID) addsetelement ?items and
?temp <- cardinality ?items and
?new_length <- ?temp + 1.
...
```

B. Animation of Models

X-System is a tool created to support modelling with X-machines [9]. Using XMDL as the modelling language, X-System allows the animation of X-Machine models. The parser of XMDL was built using the DCG (Definite Clause Grammars) notation, which is integrated in the Prolog language and is responsible for the syntax check of the models as well as type and logical errors. After the parser has ensured the correctness and completeness of a model, X-System allows its compilation into Prolog executable code. The Prolog code may then be used by X-System's animator, a program which implements an algorithm that simulates the computation of an X-machine. Through this simulation it is possible first of all for the developers to informally verify that the model simulates the actual system under development, and then also to demonstrate the model to the domain experts aiding them to identify any misconceptions regarding the user requirements between them and the development team.

V. $\mathcal{X}m$CTL Model Checking

An automatic and formal verification technique for X-machines based on model checking is provided. This formal verification technique for X-machine models enables the designer to verify the developed model against temporal logic formulas that express the properties that the system should have. For this purpose an extended version of temporal logic was devised that is appropriate for X-machine models, named $\mathcal{X}m$CTL [4].

The version of temporal logic that is usually used to express the properties in model checking is the Computation Tree Logic (CTL). In CTL [12] each of the five temporal operators ($\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$, $\mathbf{U}$, $\mathbf{R}$) must be preceded by either $\mathbf{A}$ (for all paths) or $\mathbf{E}$ (there exists path) path quantifiers. The temporal operators used in $\mathcal{X}m$CTL are the operators of CTL with the addition of two new memory quantifiers, namely $\mathbf{M_x}$ and $\mathbf{m_x}$:

- $\mathbf{M_x}$ (for all memory instances) requires that a property holds at all possible memory instances of an X-machine state.
- $\mathbf{m_x}$ (there exists a memory instance) requires that a property holds at some memory instances of an X-machine state.

Having developed an X-machine model type it is possible to verify it for desired properties. The properties are expressed as $\mathcal{X}m$CTL formulas, which together with the X-machine model is given as input to the model checker. This tool outputs true if the model satisfies the property or false together with a counterexample. If the latter is the outcome the model is altered accordingly using the debugging information (counterexample) until the model will satisfy the property. When all formulas have been verified the X-machine model is proved to have all the desired properties, i.e. the model is "correct" with respect to the requirements.

For example in the case of the buffer model the property the number of elements in the sequence will never exceed buffer's capacity can be expressed with the following $\mathcal{X}m$CTL formula: $\mathbf{AG}\,\mathbf{M_x}(card(M(1)) \leq M(2))$. The notation $M(i)$ is used in $\mathcal{X}m$CTL to denote the i-th variable in the memory. The formula can be interpreted as: for all computational paths of the X-machine model and for all states in these paths the cardinality of the sequence holding the items will be always less or equal to the capacity of the buffer for all memory instances of each state.

VI. Testing

One important advantage that modelling with X-machines has to offer is the fact that it allows for complete testing of the models. The devised testing strategy for X-machine models was proved to find all faults in an implementation [13] and it is a generalisation of Chow's W-method for the testing of FSMs[14]. The method works based on certain assumptions, and design-for-test conditions, i.e. output distinguishability and completeness, and can produce a complete test set of input sequences. In order to check whether the design-for-test conditions are met, the executable model described above is used by providing a transition cover set (S) and a characterisation set (W). Informally, a characterisation set $W \subseteq \Phi*$ is a set of processing functions for which any two distinct states of the machine are distinguishable. The state cover $S \subseteq \Phi*$ is a set of processing functions such that all states are reachable by the initial state.

In the provided example, the modeller can derive the transition cover set and a characterisation set of the processing reactive agent model: W = {`start`, `reset`, `unblock`, `end_process`}, S = {`request`, `start`, `start end_process`, `start end_process block`,`start end_process block unblock` }

Consequently, the complete test case set is produced by applying the test case function [6] and indicatively a test case is:

```
test case 1
input sequence: (typeA, 1) (finish, process)
           (out_buffer, full) (out_buffer, not_full)
output sequence: process_started processing_finished
                 agent_block agent_unblock
```

VII. Communication of agents

A. Theory of Communicating X-machines

A Communicating X-machine System Z as defined in [8] is a 2-tuple:

$$Z = ((C_i)_{i=1,\ldots,n}, CR)$$

where:

- C_i is the i-th Communicating X-machine agent, and
- CR is a relation defining the communication among the agents, $CR \subseteq C \times C$ and $C = \{C_1, \ldots, C_n\}$. A tuple $(C_i, C_k) \in CR$ denotes that the X-machine agent C_i can output a message to a corresponding input stream of X-machine agent C_k for any $i, k \in \{1, \ldots, n\}$, $i \neq k$.

Communicating X-machine model consists of several X-machine agents that are able to interact by exchanging messages. The structure CR defines a directed graph which statically determines the direction of messages between agents. An X-machine agent is defined as an X-machine, i.e. X-machine type with initial memory and initial state, in which the functions do not only read and write from/to their input and output streams respectively but also read and write from/to streams that are used to communicate with other X-machine agents. More analytically, functions are of the form: $\varphi_i\ ((\sigma)_j, m) = ((\gamma)_k, m')$ where $(\sigma)_j$ means that input is provided by machine C_j whereas $(\gamma)_k$ denotes an outgoing message to machine C_k. If $i = j$ and/or $i = k$, that means that machine C_i reads from its standard input stream and/or writes to its standard output stream.

Graphically on the state transition diagram we denote the acceptance of input by a stream other than the standard by a solid circle along with the name C_j of the communicating X-machine agent that sends it. Similarly, a solid diamond with the name C_k denotes that output is sent to the C_k communicating X-machine agent.

B. Creation of Communicating Agents with XMDL-c

XMDL has also been extended (XMDL-c) in order to code communicating agents. XMDL-c is used to define instances of models by providing a new initial state and a new initial memory instance:

```
#model <model_instance> instance_of <model_type>
with:
#init_state <initial_state>;
#init_memory <initial_memory>.
```

In addition, XMDL-c provides syntax that facilitates the definition of the communicating functions. The general syntax is the following:

```
#communication of function <function_name>:
#reads from <model instance>;
#writes <message tuple> to <model_instance>
    using <variable> from output <output tuple> and
    using <variable> from input <input tuple> and
    using <variable> from memory <memory tuple>
where   <informative expression>.
```

A function can either read or write or both from other agents (model instances). It is not necessary to specify the incoming message because it is of the same type as the input defined in the original agent. However, it is necessary to specify the outgoing message as a tuple which may contain values that exist in either output or input tuples of the function or even in the memory tuple of the agent. The informative expression is used to perform various operations on these values before they become part of the outgoing message tuple.

C. Compiling X-machine agents

CommX-System is a tool created to support modelling with Communicating X-machines [8]. To start with, CommX-System initially uses an XMDL-c description of the communication interface of a system's agent. The parser ensures the syntactic and logical correctness of the description, the compiler performs the semantic analysis and transforms the description into executable code. The compiler is then responsible for combining the communication code with that of the actual model code. One unique executable file is produced corresponding to the communicating agent of the overall system. After all the above have been performed for each of the agents of the system, all produced files are combined to create one that corresponds to the entire system and which will be used by the tool's animator.

Indicatively, we present a part of the communicating system. According to the description of the problem, the input buffer and the processing reactive agent should communicate in the sense that an item should be sent from the buffer to the reactive agent when the reactive agent is ready to process it. The following XMDL-c code creates a agent for the (input) buffer and its communication with the reactive agent agent.

```
#model buf_in instance_of buffer
with:
#init_state {empty};
#init_memory (nil, 5).

#communication of function remove_item:
#writes ((?item,?id)) to (mach)
using ?item from output (?m, (?item,?id)) and
using ?id from output (?m, (?item,?id)).

#communication of function become_empty:
#writes ((?item,?id)) to (mach)
using ?item from output (?m, (?item,?id)) and
using ?id from output (?m, (?item,?id)).
```

Similarly, the following XMDL-c code defines the reactive agent model and its communication with the input buffer. The communication between these two agent X-machine models is depicted in figure 4. The reactive agent and the output buffer interact in a similar way.

```
#model robot1 instance_of reactive agent
with:
#init_state {ready};
#init_memory ((none,0)).

#communication of function start:
#reads from buf_in.
#end.
```

VIII. The Overall System

So far, we have followed steps (a) to (e) of the methodology and we assume that all agents (the input buffer, the processing reactive agent and the output buffer) have been developed, animated, verified and tested as well as communication between them has been established.

The system modelled at this stage is a simple agent system with one reactive agent that has an input and an output buffer as depicted in figure 2 without the controller agents. The input buffer stores the items added in it and communicates to the reactive agent an item stored in the buffer, whenever the reactive agent needs one; i.e. the input buffer models a heap. In a similar fashion, the output buffer, accepts an item processed by the reactive agent and stores it. In the case that it is required to employ an input buffer with another discipline (e.g. FIFO), then it is necessary to create another buffer agent that has the same transition diagram as the one shown in figure 4, but different implementation of the transition function `remove_item`. It is therefore evident that this approach is not coping well with changes mainly because it restricts reusability.

The last step (f) demonstrates the flexibility of the proposed methodology addressing this issue. Changes in the system behaviour can be easily handled by the proposed methodology with the addition of (probably off-the-self) controller agents that encapsulate the desired behaviour. In the manufacturing facility case the input controller (ctrl-in) has been added to control the feeding of the items from the input buffer to the reactive agent in a FIFO manner. The complete model is depicted in figure 5 providing a flexible and modular solution. Any change of the requirements e.g. in the manner the items are fed to the reactive agent can be dealt with the use of a different specialised input controller, replacing the old one in the model.

It has been demonstrated that the proposed methodology offers an intuitive way to model agent-based systems by providing the flexibility that each agent identified in the real system is mapped directly to an X-machine agent model in the design of the system. By applying this methodology to agent-based systems it is possible to incrementally model the complete system and assure that all desired properties of the agents of the system hold in the final product.

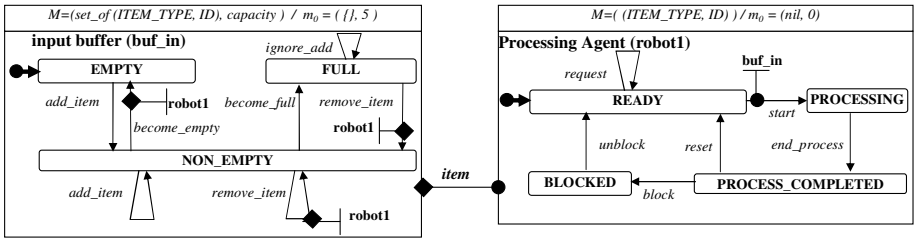

Fig. 4. Communication between an instance of a Processing Agent and an Input Buffer

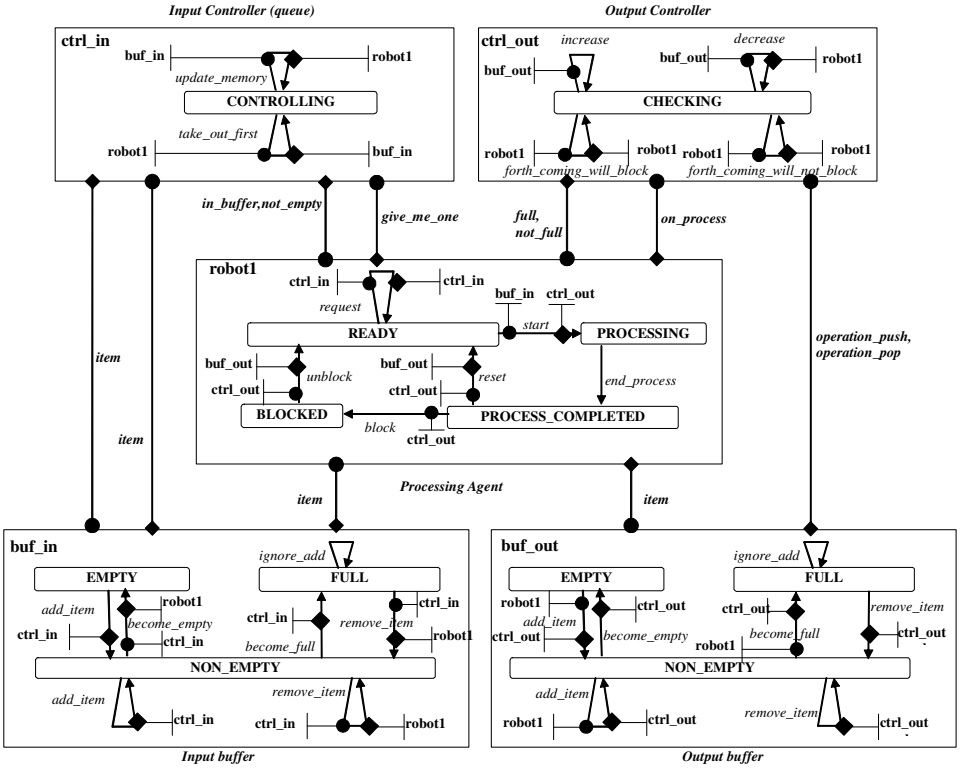

Fig. 5. The model of the complete system

IX. Conclusions and Further Work

We have presented a methodology for developing simple reactive agent-based systems using communicating X-machines formal method. X-machines attracted many researchers interest over the last twenty years [15] mainly because of the intuitiveness in modelling reactive systems and the additional features they provide in terms of testing and verification. The main contribution of this paper is the detailed description of the methodology that allows to scale up to larger and more complex systems with a focus on developing correct components, the formal introduction of the X-machine type models and their instances as parts of a more complex system and the simple but general and complete example to demonstrate the applicability of the proposed method. The methodology and its accompanying tools impose an incremental bottom-up practical development. It is useful in cases where complex systems can be viewed as an aggregation of simple agents that can communicate in order to achieve the overall behaviour of a distributed system. A particular example is the multi-agent systems [3], in which similar methodologies might be employed, such as Gaia, AAII, Cassiopeia etc. [1].

With the continuous verification and testing of agents from the early stages risks are reduced and the developer is confident of the correctness of the system under development throughout the whole process. It is worth noticing that the proposed methodology utilises a priori techniques to avoid any flaws in the early stages of the development together with a posteriori techniques to discover any undiscovered flaws in later stages. This way it makes the best use of the development effort to achieve highest confidence in the quality of the developed agents that have been verified and tested therefore they can be reused as trusted agents. The proposed communicating X-machine concept is based on the idea of reusability, thus minimizing the development time without risking the quality of

the product. Further works also include modelling of dynamic systems, models of which the configuration of communicating agents changes over time. A set of appropriate rules has been devised [16], [11] and ideas are borrowed from biological computational paradigms, such as membrane computing [17], in order to facilitate definition of the appropriate hybrid formal method [18].

There is a need for the extension of the model checking technique in order to facilitate the formal verification of communicating X-machine models. Research is conducted towards this direction but also towards the establishment of a successful testing strategy for the communicating X-machine models [19]. Finally there is a continuous need to evaluate the proposed methodology with real case studies and with industrial development teams to prove and not demonstrate its applicability and to compare with other industrial strength methodologies.

REFERENCES

[1] M. Wooldridge and P. Ciancarini, "Agent-Oriented Software Engineering: The State of the Art," in *Agent-Oriented Software Engineering*, ser. Lecture Notes in AI, P. Ciancarini and M. Wooldridge, Eds. Springer-Verlag, 2001, vol. 1957, pp. 337–350.

[2] B. Meyer, "The Grand Challenge of Trusted Components," in *25th International Conference on Software Engineering*, Portland, Oregon, May 2003, pp. 660–667.

[3] P. Kefalas, M. Holcombe, G. Eleftherakis, and M. Gheorghe, "A formal method for the development of agent-based systems," in *Intelligent Agent Software Engineering*, V. Plekhanova, Ed. Idea Group Publishing, 2003, ch. 4, pp. 68–98.

[4] G. Eleftherakis and P. Kefalas, "Formal Verification of Generalised State Machines," in *12th Panhellenic Conference on Informatics (PCI 2008)*, S.Gritzalis, D.Plexousakis, and D.Pnevmatikatos, Eds. Samos, Greece: IEEE Computer Society, Aug. 2008, pp. 227–231.

[5] F. Ipate, "Complete deterministic stream X-machine testing," *Formal Aspects of Computing*, vol. 16, no. 4, pp. 374–386, Nov. 2004.

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

[7] F. Ipate and M. Holcombe, "An integration testing method that is proved to find all faults," *International Journal of Computer Mathematics*, vol. 63, no. 3, pp. 159–178, 1997.

[8] P. Kefalas, G. Eleftherakis, and E. Kehris, "Communicating X-machines: a practical approach for formal and modular specification of large systems," *Information and Software Technology*, vol. 45, no. 5, pp. 269–280, Apr. 2003, woS Impact Factor (1.821 - 1.426/5y).

[9] P. Kefalas, G. Eleftherakis, and A. Sotiriadou, "Developing Tools for Formal Methods," in *9th Panhellenic Conference on Informatics*, Thessaloniki, Nov. 2003, pp. 625–639.

[10] G. Eleftherakis, P. Kefalas, A. Sotiriadou, and E. Kehris, "Modeling Biology Inspired Reactive Agents Using X-machines," *Proceedings of World Academy of Science, Engineering and Technology*, vol. 1, pp. 93–96, Jan. 2005, also published in:International Conference on Computational Intelligence (ICCI04) [c12].

[11] P. Kefalas, I. Stamatopoulou, I. Sakellariou, and G. Eleftherakis, "Transforming Communicating X-machines into P Systems," *Natural Computing*, vol. 8, no. 4, pp. 817–832, Dec. 2009.

[12] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, Massachusetts: MIT Press, 1999.

[13] F. Ipate and M. Holcombe, "Specification and testing using generalised machines: a presentation and a case study," *Software Testing, Verification and Reliability*, vol. 8, pp. 61–81, 1998.

[14] T. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp. 178–187, 1978.

[15] "Special Issue on X-machines," *Formal Aspects of Computing*, vol. 12, no. 6, 2000.

[16] P. Kefalas, G. Eleftherakis, M. Holcombe, and I. Stamatopoulou, "Formal modelling of the dynamic behaviour of biology-inspired agent-based systems," in *Molecular Computational Models: Unconventional Approaches*, M. Gheorghe, Ed. Idea Group Publishing, 2005, ch. 9, pp. 243–276.

[17] P. Kefalas, I. Stamatopoulou, M. Gheorghe, and G. Eleftherakis, "Membrane Computing and X-machines," in *The Oxford Handbook of Membrane Computing*, G. Paun, Ed. Oxford University Press, 2009, ch. 23.4, pp. 612–620.

[18] I. Stamatopoulou, P. Kefalas, and M. Gheorghe, "Operas: A framework for the formal modelling of multi-agent systems and its application to swarm-based systems," in *ESAW*, ser. Lecture Notes in Computer Science, A. Artikis, G. M. P. O'Hare, K. Stathis, and G. A. Vouros, Eds., vol. 4995. Springer, 2007, pp. 158–174.

[19] F. Ipate, T. Bălănescu, and G. Eleftherakis, "Testing Communicating Stream X-machines," in *1st Balkan Conference on Informatics*, Thessaloniki, Nov. 2003, pp. 161–173.