

A FORMAL METHOD FOR THE DEVELOPMENT OF AGENT-BASED SYSTEMS

P. Kefalas¹, M. Holcombe², G.Eleftherakis¹, M.Gheorghe²

¹ Dept. of Computer Science, CITY College,
13 Tsimiski Street, Thessaloniki 546 24, Greece
Tel. +30310-275575, Fax. +30310-287564
{kefalas, eleftherakis}@city.academic.gr

² Dept. of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK
{m.holcombe, m.gheorghe}@dcs.shef.ac.uk

A FORMAL METHOD FOR THE DEVELOPMENT OF AGENT-BASED SYSTEMS

ABSTRACT

Recent advances in both the testing and verification of software based on formal specifications of the system to be built have reached a point where the ideas can be applied in a powerful way in the design of agent-based systems. The software engineering research has highlighted a number of important issues: the importance of the type of modelling technique used; the careful design of the model to enable powerful testing techniques to be used; the automated verification of the behavioural properties of the system; the need to provide a mechanism for translating the formal models into executable software in a simple and transparent way.

This chapter presents a detailed and comprehensive account of the ways in which some modern software engineering research can be applied to the construction of effective, and reliable agent-based software systems. More specifically, we intend to show how simple agents motivated from biology can be modelled as X-machines. Such modelling will facilitate both verification and testing of an agent model, since appropriate strategies for model checking and testing are already developed around the X-machine method. In addition, modular construction of agent models is feasible since X-machines are provided with communicating features, which allow simple models to interact.

INTRODUCTION

An *agent* is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives (Jennings, 2000). There are two fundamental concepts associated with any dynamic or reactive system, such as an agent, that is situated in and reacting with some environment (Holcombe & Ipaté, 1998):

- the environment itself, which could be precisely or ill-specified or even completely unknown, but nevertheless involves identifying the important aspects of the environment and the way in which they may change in accordance with the activities of the agent,
- the agent will be responding to environmental changes by changing its basic parameters and possibly affecting the environment as well. Thus, there are two ways in which the agent reacts, i.e. it undergoes internal changes and it produces outputs that affect the environment.

Agents, as highly dynamic systems, are concerned with three essential factors:

- a set of appropriate environmental stimuli or inputs,
- a set of internal states of the agent, and
- a rule that relates the two above and determines what the agent state will change to if a particular input arrives while the agent is in a particular state.

One of the challenges that emerge in intelligent agent engineering is to develop agent models and agent implementations that are “*correct*”. According to Holcombe & Ipaté (1998), the criteria for “correctness” are:

- the initial agent model should match with the requirements,
- the agent model should satisfy any necessary properties in order to meet its design objectives, and

- the implementation should pass all tests constructed using a complete functional test generation method.

All the above criteria are closely related to three stages of agent system development, i.e. *modelling*, *verification* and *testing*.

Although agent-oriented software engineering aims to manage the inherent complexity of software systems (Wooldridge & Ciancarini, 2001), there is still no evidence to suggest that any methodology proposed leads towards “correct” systems. In the last few decades, there has been a strong debate on whether *formal methods* can achieve this goal. Academics and practitioners adopted extreme positions either for or against formal methods (Young, 1991). It is, however, apparent that the truth lies somewhere between and that there is a need for use of formal methods in software engineering in general (Clarke & Wing, 1996), while there are several specific cases proving the applicability of formal methods in agent development, as we shall see in the next section.

Software system specification has centred on the use of models of data types, either functional or relational models such as *Z* (Spivey, 1989) or *VDM* (Jones, 1990) or axiomatic ones such as *OBJ* (Futatsugi et al., 1985). 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* (Wulf et al., 1981) or *Petri Nets* (Reisig, 1985) capture the essential feature, which is “change”, 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* (Harel 1987), 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. This chapter will introduce such a formal method, namely X-machines, which closely suits the needs of agent development, while at the same time being intuitive and practical.

FORMAL METHODS IN AGENT-BASED SYSTEMS

In agent oriented engineering, there have been several attempts to use formal methods, each one focusing on different aspects of agent systems development. One of them was to formalise *PRS* (Procedural Reasoning System), a variant of the *BDI* architecture (Rao & Georgeff, 1995) with the use of *Z*, in order to understand the architecture in a better way, to be able to move to the implementation through refinement of the specification and to be able to develop proof theories for the architecture (D’Iverno et al. 1998). Trying to capture the dynamics of an agent system, Rosenschein & Kaebling (1995) viewed an agent as a situated automaton that generates a mapping from inputs to outputs, mediated by its internal state. Brazier et al. (1995) developed the *DESIRE* framework, which focuses on the specification of the dynamics of the reasoning and acting behaviour of multi-agent systems. In an attempt to verify whether properties of agent models are true, work has been done on model checking of multi-agent systems with re-use of existing technology and tools (Benerecetti et al. 1999, Rao & Georgeff, 1993). Towards implementation of agent systems, Attoui & Hasbani (1997) focused on program generation of reactive systems through a formal transformation process. A wider approach is taken by Fisher & Wooldridge (1997) who utilise *Concurrent METATEM* in order to formally specify multi-agent systems and then directly execute the specification while verifying important temporal properties of the system. Finally, in a less

formal approach, extensions to *UML* to accommodate the distinctive requirements of agents (*AUML*) were proposed (Odell et al, 2000).

In this chapter, we intend to show how simple agents motivated from biology can be modelled as X-machines. Such modelling will facilitate both verification and testing of an agent model, since appropriate strategies for model checking and testing are already developed around the X-machine method. In addition, modular construction of agent models is feasible since X-machines are provided with communicating features, which allow simple models to interact. Finally, tools developed for X-machines are briefly presented in order to demonstrate the practicality of the approach.

X-MACHINE DEFINITION

A *X-machine* is a general computational machine introduced by Eilenberg (1974) and extended by Holcombe (1988) that resembles a Finite State Machine (FSM) but with two significant differences:

- there is memory attached to the machine, and
- 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. Other machine models like pushdown automata or *Turing machines* are too low level and hence of little use for specification of real systems. X-machines employ a diagrammatic approach of modelling the control by extending the expressive power of the FSM. They are capable of modelling both the data and the control of a system. Data is held in memory, which is attached to the X-machine. Transitions between states are performed through the application of functions, which are written in a formal notation and model the processing of the data. 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 (Ipate & Holcombe, 1998) is an 8-tuple $XM = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, 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 XM , 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 state transition diagram, $F: Q \times \Phi \rightarrow Q$
- q_0 and m_0 are the initial state and memory respectively.

X-machines can be used as a core method for an integrated formal methodology of developing correct systems.

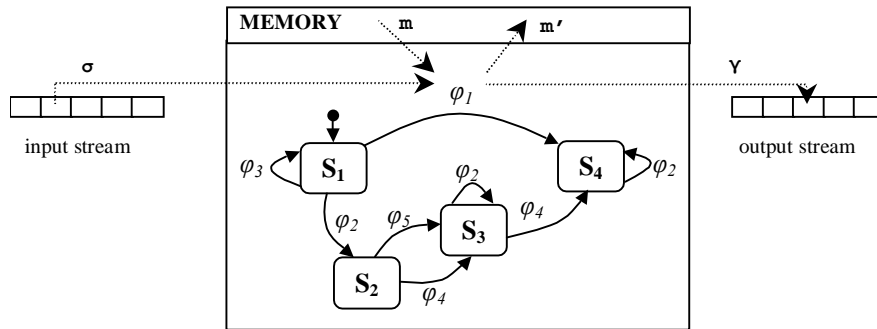


Fig. 1. An abstract example of a X-machine; φ_i : functions operating on inputs and memory, S_i : states. The general formal of functions is: $\varphi(\sigma, m) = (\gamma, m')$ if condition

The X-machine integrates both the control and data processing while allowing them to be described separately. The X-machine formal method forms the basis for a specification/modelling language with a great potential value to software engineers. It is rather intuitive, while at the same time formal descriptions of data types and functions can be written in any known mathematical notation. Finally, X-machines can be extended by adding new features to the original model, such as hierarchical decomposition and communication, which will be described later. Such features are particularly interesting in agent-based systems.

FORMAL AGENT MODELLING

Many biological processes seem to behave like agents, as for example a colony of ants. Much research has been based on such behaviour in order to solve interesting problems (Dorigo & Di Caro, 1999). An important task of some ants is to find food and carry it to their nest. This can be accomplished by searching for food at random or by following pheromone trails that other ants have left on their return back to the nest (Deneubourg et al., 1990). While moving, an ant should avoid obstacles. Once food is found, an ant should leave a pheromone trail while travelling back to its nest, thus implicitly communicating with other ants the destination of a source where food may be found. When the nest is found, the ant drops the food. Clearly, this is a reactive agent that receives inputs from the environment and acts upon these inputs according to the state in which the agent is. Such reactive agents can be fairly easily modelled by a FSM in a rather straightforward way by specifying the states and the inputs (percepts) to be used for state transitions (Fig.2).

The FSM lacks the ability to model any non-trivial data structures. In more complex tasks, one can imagine that the actions of the agents will also be determined by the values stored in its memory. For example, an agent may know its position, remember the position of the food source or the position of obstacles, thus building a map of the environment in order to make the task eventually more efficient. Using FSM or variants of it (Brooks 1986, Rosenschein & Kaebbling 1995) for such agents is rather complicated since the number of states increases in combinatorial fashion to the possible values of the memory structure. X-machines can facilitate modelling of agents that demand remembering as well as reactivity. Fig.3 shows the model of an ant that searches for food, but also remembers food positions in order to set up its next goals. The behaviour of obstacle avoidance is omitted for simplicity.

where *none* indicates that no food is carried.

The initial memory and the initial states are respectively:

$$m_0 = (none, (0,0), nil)$$

$$q_0 = \text{"At Nest"}$$

It is assumed that the nest is at position (0,0). The next state partial function is depicted with the state diagram in Fig.3. The type Φ is a set of functions of the form:

$$function_name(input_tuple, memory_tuple) \rightarrow (output, memory_tuple'), \text{ if condition.}$$

For example, here are some function definitions:

$$lift_food(f, (x,y), (none, (x,y), foodlist)) \rightarrow$$

$$("lifting\ food", (f, (x,y), <(x,y) :: foodlist>)), \quad \text{if } f \in FOOD \wedge (x,y) \notin foodlist$$

$$find_food(f, (fpx, fpy), (food, (x,y), foodlist)) \rightarrow$$

$$("more\ food", (food, (x,y), <(fpx, fpy) :: foodlist>)), \quad \text{if } f \in FOOD \wedge f \notin foodlist$$

$$drop_food((nest, 0,0), (food, (x,y), foodlist)) \rightarrow$$

$$("dropping\ food", (none, (0,0), foodlist))$$

$$find_nest((nest, 0,0), (none, (x,y), foodlist)) \rightarrow$$

$$("found\ nest\ again", (none, (0,0), foodlist))$$

VERIFICATION OF AGENT MODELS

Having designed a model for an agent, it would be desirable to verify whether it corresponds to the requirements, i.e. at all circumstances during the existence of the agent modelled in some way, its required properties are true in that model. *Model checking* is a formal verification technique, which determines whether given properties of a system are satisfied by a model. A model checker takes a model and a property as inputs and outputs either a claim that the property is true or a counterexample falsifying the property. In order to use model checking, the most efficient way to express a model is any kind of state machine, a *CCS* agent, a *Petri Net*, a *CSP* agent, etc. The most common properties to check are either something will never occur or something will eventually occur. In *Temporal Logic Model Checking* (Clarke et al., 1986) a property is expressed as a formula in a certain temporal logic. The verification can be accomplished using an efficient breadth first search procedure, which views the transition system as a model for the logic and determines if the specifications are satisfied by that model. This approach is simple, completely automated but has one major problem, namely the state explosion. The latter can be handled to some extent by a model checking variant, the *symbolic model checking* (McMillan, 1993). Model checking is sometimes proved to be more efficient than other formal verification techniques, such as theorem proving (Burch et al., 1992). On the other hand, model checking is formal as opposed to simulation, which may reveal inconsistencies and misconceptions in a model, but does not guarantee completeness of the model with respect to requirements.

Bearing in mind the most usual definition of a model, i.e. a labelled state transition graph, also called a *Kripke structure* (Kripke, 1963), model checking utilises algorithms that, given a temporal logic formula, verifies whether properties hold in the model. A Kripke structure is a tuple $\langle Q, R, L \rangle$ where:

- Q is non-empty set of states,
- R is a binary relation on Q , i.e. $R \subseteq Q \times Q$, which shows which states are related to other states, and
- $L: Q \rightarrow 2^{Prop}$ is a truth assignment function that shows which propositions are true in each state, where $Prop$ is a set of atomic propositions.

Temporal logic formulae, e.g. *CTL** formulae, are constructed through the use of operators combined with the path quantifiers *A* (meaning “for all paths”) or *E* (meaning “there exists a path”). The five basic *CTL** operators are (Emerson & Halpern, 1986):

- *X* (next time) requires that a property holds in the next state,
- *F* (eventually) requires a property to hold at some state on a path,
- *G* (always) requires a property to hold at every state on a path,
- *U* (until) requires a property *p* to hold in a path until another property *q* holds,
- *R* (release) as a dual operator of *U*.

The above logic could be extended to accommodate past, present and future properties of an agent system, as in *FML* (Fisher & Wooldridge, 1997).

Having constructed a model of an agent as a X-machine, it is possible to apply existing model checking techniques to verify its properties. That would require transformation of a X-machine into another model that resembles a Kripke structure. Such a process exists, called the exhaustive refinement of a X-machine to a FSM, and results in a model in which *CTL** formula may be applied. However, exhaustive refinement suffers two major disadvantages:

- the loss of expressiveness that the X-machine possesses, and
- the combinatorial explosion.

The former has to do with the memory structure attached to X-machine. In an equivalent FSM resulting from the process of refinement, the memory will be implicitly contained in states. It would be therefore impossible to verify that certain properties are true for some memory instances of the states of the original X-machine model, since this information is lost during the refinement process. The latter has to do with properties which are contained in the model’s memory but do not play any role in model checking with respect some other properties. Exhaustive refinement may result in a possibly infinite state space, if such properties are included in the equivalent FSM, thus making model checking impossible.

In order to apply model checking in X-machines, temporal logic is extended with memory quantifier operators:

- M_x , for all memory instances and
- m_x , there exist memory instances,

which together with the basic temporal operators of *CTL**, can form expressions suitable for checking the properties of an agent model. The resulting logic *XmCTL* can verify the model expressed as X-machine against the requirements, since it can prove that certain properties, which implicitly reside in the memory of X-machine are true (Eleftherakis & Kefalas, 2001).

For example, in an agent whose task is to carry food to its nest as in the example of Fig.3, model checking can verify whether eventually food will be dropped in the nest by the formula:

$$AG [\neg M_x (m_1 \neq \text{none}) \vee EFM_x (m_1 = \text{none})]$$

where m_1 indicates the first element of the memory tuple. The formula states that, in all states of the X-machine, it is true that either the ant does not hold any food or there exists a path after that state where eventually the ant does not hold any food.

Another example is the formula:

$$E [M_x (m_1 = \text{none}) U M_x (m_1 \neq \text{none})]$$

i.e., there exists a path in which the ant eventually holds food and in all previous states the ant holds nothing. Also, another useful property to be checked is:

$$\neg EFM_x [(m_1 \neq \text{none}) \wedge (m_3 = \text{nil})]$$

i.e., if the ant holds something then the food list is not empty.

The new syntax and semantics facilitate model checking of X-machines in two ways:

- expressiveness suited to the X-machine model, and

- effective reduction of the state space through selective refinement of the original X-machine model.

COMPLETE TESTING OF AGENTS

In the previous section, we have focused in modelling of an agent and verification of the models specified with respect to requirements. Having ensured that the model is “correct”, we need to also ensure that the implementation is “correct”, this time with respect to the model. This can be achieved through testing, but only under one important assumption, i.e. testing is complete. To guarantee “correctness” of the implementation, one must be certain that all tests are performed, and the results correspond to what the model has specified.

Holcombe & Ipate (1998) presented a testing method, which is a generalisation of Chow’s *W-method* (Chow, 1978) for FSM testing. It is proved that this testing method finds all faults in the implementation (Ipate & Holcombe, 1998). The method works based on the following assumptions:

- the specification and the implementation of the system can be represented as X-machines,
- the X-machine corresponding to the specification and the X-machine corresponding to the implementation have the same type Φ .

Assuming the above, the method also requires that:

- the X-machine satisfies the design for test conditions, and
- its associated automaton is minimal.

The *associated automaton* of a X-machine is the conversion of the X-machine to a FSM by treating the elements of Φ as abstract input symbols. The design for test conditions states that the type Φ of the two machines is both *complete* with respect to memory and *output distinguishable*.

A processing function $\varphi \in \Phi$ is called *complete* with respect to memory if:

$$\forall m \in M, \exists \sigma \in \Sigma \text{ such that } (m, \sigma) \in \text{dom } \varphi$$

A type Φ is called *complete* with respect to memory M , if any basic function will be able to process all memory values, that is if:

$$\forall \varphi \in \Phi, \varphi \text{ is complete with respect to } M$$

A type Φ is called *output distinguishable* if any two different processing functions will produce different outputs on each memory/input pair, that is if:

$$\forall \varphi_1, \varphi_2 \in \Phi \text{ if } \exists m \in M, \sigma \in \Sigma \text{ such that for some } m_1', m_2' \in M, \gamma \in \Gamma \\ \varphi_1(m, \sigma) = (\gamma, m_1') \text{ and } \varphi_2(m, \sigma) = (\gamma, m_2'), \text{ then } \varphi_1 \neq \varphi_2.$$

If Φ is not complete then additional input symbols may be introduced such as to make processing functions complete (Holcombe & Ipate, 1998).

In Fig.4, the X-machine illustrates a model of an ant that either looks for food at random or follows the pheromone trail to find food and nest. The input set is $\Sigma = \{\text{space, nest, pheromone, food}\}$. The X-machine satisfies the design for test conditions and its associated automaton is minimal.

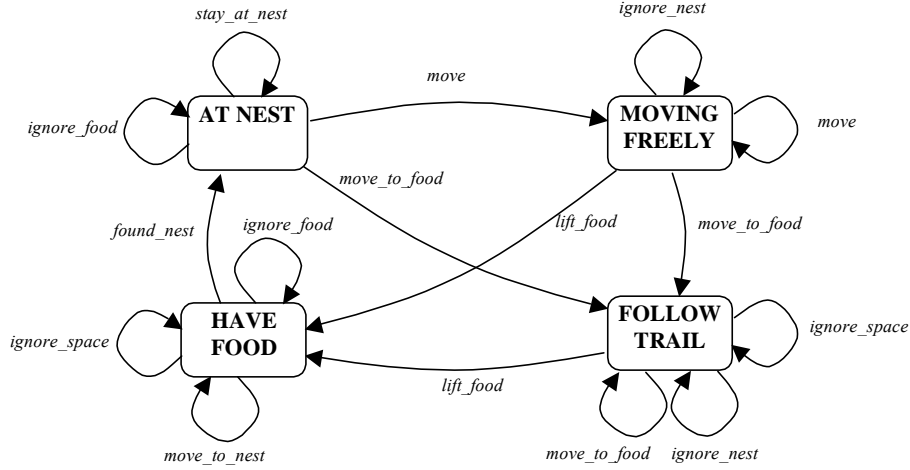


Fig. 4. An X-machine that satisfies the design for test conditions

When these requirements are met, the W-method may be employed to produce the k -test set X of the associated automaton, where k is the difference of the number of states of the two associated FSMs. The test-set X consists of processing functions for the associated automaton, and it is given by the formula:

$$X = S(\Phi^{k+1} \cup \Phi^k \cup \dots \cup \Phi \cup \{\varepsilon\})W$$

where W is a *characterisation set* and S a *state cover*. 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 q_0 . The W and S sets in the agent X-machine in Fig.4 are:

$$W = [\text{stay_at_nest}, \text{move move_to_food}, \text{found_nest}]$$

$$S = [\varepsilon, \text{move}, \text{move move_to_food}, \text{move_to_food lift_food}]$$

The derived test-set X , for $k=0$, i.e. model and implementation are considered as FSM with the same number of states, is the following:

$$X = \{ \text{move move move_to_food}, \text{move move move move_to_food}, \\ \text{move ignore_nest move move_to_food}, \text{move lift_food found_nest}, \\ \text{move move_to_food lift_food found_nest}, \text{move_to_food lift_food found_nest}, \\ \text{move_to_food lift_food ignore_space found_nest}, \dots \}$$

The fundamental test function is defined recursively, and converts these sequences into sequences of inputs of the X-machine. Let $XM = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ be a deterministic stream X-machine with Φ complete with respect to M and let $q \in Q, m \in M$. A function $t_{q,m}: \Phi^* \rightarrow \Sigma^*$ will be called a test function of M with respect to q and m , will be defined recursively as (Ipaté and Holcombe, 1998):

$$t_{q,m}(\varepsilon) = \varepsilon \text{ (the empty input symbol)}$$

or

$$t_{q,m}(\varphi_1 \dots \varphi_{n+1}) = \begin{cases} t_{q,m}(\varphi_1 \varphi_2 \dots \varphi_n) \sigma_{n+1}, \\ \text{if } \exists \text{ a path } q, q_1, \dots, q_{n-1}, q_n \text{ in } M \text{ starting from } q, \text{ where } \sigma_{n+1} \text{ is} \\ \text{such that } (m_n, \sigma_{n+1}) \in \text{dom } \varphi_{n+1} \text{ and } m_n \text{ is the final memory} \\ \text{value computed by the machine along the above path on the} \\ \text{input sequence } t_{q,m}(\varphi_1 \varphi_2 \dots \varphi_n) \\ t_{q,m}(\varphi_1 \varphi_2 \dots \varphi_n), \text{ otherwise} \end{cases}$$

The test-set containing sequences of inputs for the ant X-machine is the following:

*{space space pheromone, space space space, pheromone, space nest space pheromone,
space food nest, space pheromone food nest, pheromone food nest, pheromone food
space nest, ...}*

The test-set so produced is proved to find all faults in the agent implementation. The testing process can therefore be performed automatically by checking whether the output sequences produced by the implementation are identical with the ones expected from the agent model.

AGENTS AS AGGREGATION OF BEHAVIOURS

Agents can be modelled as a stand-alone (possibly complex) X-machine as shown in the previous section. However, an agent can be also viewed as a set of simpler components, which model various different behaviours of the agent. This fits with the three principles of complex agent systems: decomposition, abstraction, and organisation (Jennings, 2001). Another approach for reactive agents is described in the *subsumption architecture* (Brooks, 1991), in which behaviours can communicate with each other in order to result in a situated agent with the desired overall robust performance. Similarly, Collinot et al. (1996) developed the *Cassiopeia* method, in which agents are defined by following three steps:

- identifying the elementary behaviours that are implied by the overall task,
- identifying the relationship between elementary behaviours, and
- identifying the organisational behaviours of the system.

A methodology of building *communicating X-machines* from existing stand-alone X-machine is developed so that modelling can be split into two separate activities:

- the modelling of X-machine components, and
- the description of the communication between these components.

The 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 modelling and communication as two separate distinct activities, and
- can use existing tools for both stand-alone and communicating X-machines.

Let us discuss the above one by one. Certain approaches for building a communicating system require a brand new conceptualisation and development of a system as a whole. This approach has a major drawback, i.e. one cannot re-use existing models that have been already verified and tested for their “correctness”. Often, in agent systems, components from other agent systems are required. A desirable approach would be to conceptualise the system as a set of independent smaller models, which need to communicate with each other. Thus, one does not need to worry about the individual components, in which model checking techniques and testing are applied, but only with appropriately linking those components. This would lead to a disciplined development methodology, which implies two distinct and largely independent development activities, i.e. building models and employing communication between them. Also, this means that existing languages and tools for modelling, model checking and testing are still useful and can be further extended to support larger communicating systems.

Several theoretical approaches for communicating X-machines have been proposed (Balanesu et al. 1999, Cowling et al. 2000, Barnard 1998). In this section we will describe the one that focuses on the practical development of communicating systems but also subsumes all others (Kefalas et al., 2001). In this approach, the functions of a X-machine, if so annotated, read input from a communicating stream 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 is the solid circle (IN port) and the solid box (OUT port) to indicate that input is read from another component and output is directed to another component respectively. For example, function φ in Fig.5 accepts its input from the model $x-m1$ and writes its output to model $x-m2$. Multiple communications channel for a single X-machine may exist. Another example is a simple form of communication between two ants. Assume that one ant is responsible to notify another ant about the position of the food source. In order to achieve communication the X-machines should be modified as illustrated in Fig.6. The function $lift_food$ of the X-machine model $ant2$ becomes:

$lift_food((f,x,y), (none, (x,y), foodlist)) \rightarrow$
 $(OUT_{x-m\ ant2}(f,x,y), (f, (x,y), <(x,y) :: foodlist>)), \text{ if } f \in FOOD \wedge (x,y) \notin foodlist$
 and the function $find_food$ of X-machine model $ant1$ becomes:

$find_food(IN_{x-m\ ant1}(f, fpx, fpy), (food, (x,y), foodlist)) \rightarrow$
 $(\text{"more food"}, (food, (x,y), <(fpx, fpy) :: foodlist>)), \text{ if } f \in FOOD \wedge (fpx, fpy) \notin foodlist$

Function $find_food$ of $ant2$ may be modified accordingly to write a message to the OUT port, if needed.

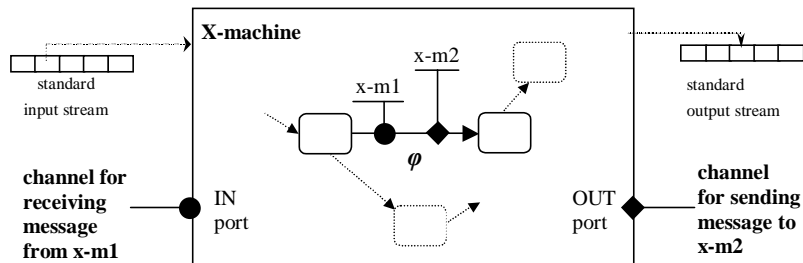


Fig. 5. An abstract example of a Communicating X-machine component.

The approach is practical, in the sense that the designer can separately model the components of an agent and then describe the way in which these components communicate. This allows a disciplined development of situated agents. Practically, as we shall see later, components can be re-used in other systems, since the only thing that needs to be changed is the communication part.

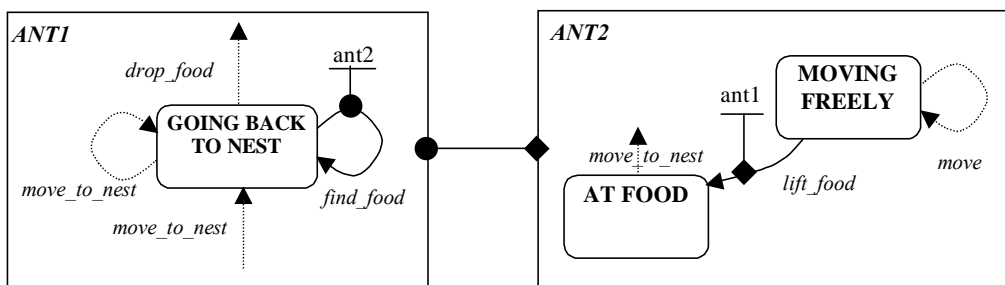


Fig. 6. Ant2 X-machine sends a message about food position in Ant1 X-machine, by utilizing a communicating port.

In the following, we will use communicating X-machines to model the collective foraging behaviour of a colony of honey-bees as it is fully compatible with the rules used by foraging honey-bees (Vries & Biesmeijer, 1998) that include specifications for:

- travelling from the nest to the source,
- searching for the source,
- collecting nectar from the source,
- travelling back to the nest,

- transmitting the information about the source (the dancing of the returning bee),
- the reaction of a bee in the nest to the dancing of a nest mate.

A foraging bee can be modelled according to a set of independent behaviours, which constitute the components of the overall agent. Fig.7 shows some of the behaviours of a foraging bee modeled as simple X-machines, with an input set Σ and a memory tuple M . Each machine has different memory, inputs (percepts), and functions. Some states and functions were on purpose named differently to show the modularity of the approach. It is assumed that the bee perceives:

- empty space to fly (*space*),
- the hive (*nest*),
- the source of nectar (*source*),
- an amount of nectar (*nectar*),
- other bees, i.e. foraging bees (*fbee*) or receiving bees (*rbee*), and finally
- understands when it has lost its orientation (*lost*).

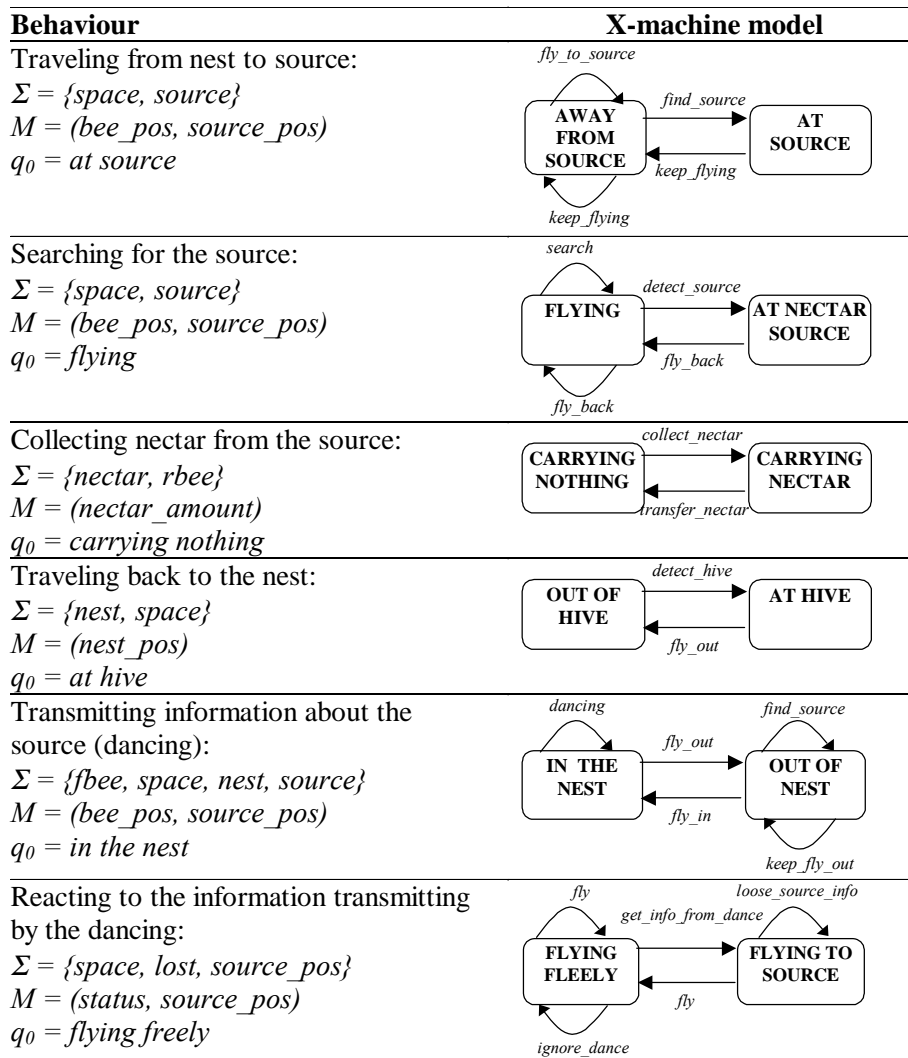


Fig. 7. The behaviours of a foraging bee modeled separately as X-machine components.

The memory of each X-machine holds information on the bee, the source and the nest positions (bee_pos , $source_pos$ and $nest_pos$), the amount of nectar carried ($nectar_amount$), and its status ($employed$ or $unemployed$). For example, consider the X-machine modelling the dancing behaviour. Its functions are defined as follows (Gheorghe et al, 2001):

$dancing(fbee, (bee_pos, source_pos)) \rightarrow ("dancing", (bee_pos, source_pos))$
 $fly_out(space, (bee_pos, source_pos)) \rightarrow ("flying out", (bee_pos', source_pos))$
 $fly_in(nest, (bee_pos, source_pos)) \rightarrow ("flying in", (bee_pos', source_pos))$
 $find_source(source, (bee_pos, source_pos)) \rightarrow ("sourcefound", (source_pos, source_pos))$
 $keep_fly_out(space, (bee_pos, source_pos)) \rightarrow ("keep flying", (bee_pos', source_pos))$

where $bee_pos, bee_pos', source_pos \in Set_of_positions$. The bee position can be calculated by some external function or some other X-machine.

Fig.8 shows in detail how communication can be achieved directly by various honey-bees, e.g. an employed foraging bee sends the source position to another foraging bee through the dancing behaviour:

$dancing(fbee, (bee_pos, source_pos)) \rightarrow$
 $(OUT_{x-m\ reacting}(source_pos), (bee_pos, source_pos))$

while an unemployed foraging bee reads the source position by the function:

$get_info_from_dance(IN_{x-m\ dancing}(source_pos), (unemployed, nil)) \rightarrow$
 $("getting source info", (employed, source_pos))$.

If the foraging bee is currently employed, it just ignores the message:

$ignore_dance(IN_{x-m\ dancing}(source_pos), (employed, source_pos)) \rightarrow$
 $("ignoring source info", (employed, source_pos))$.

The same communication takes place when a foraging bee transfers the amount of nectar that is carrying to a receiving bee waiting at the hive.

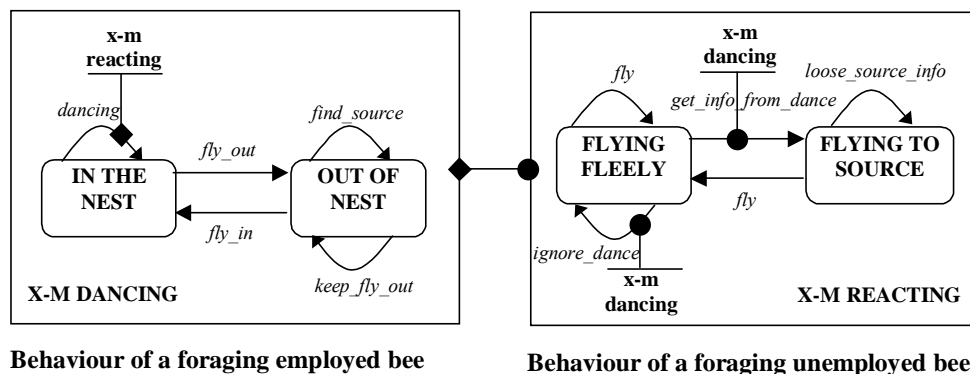
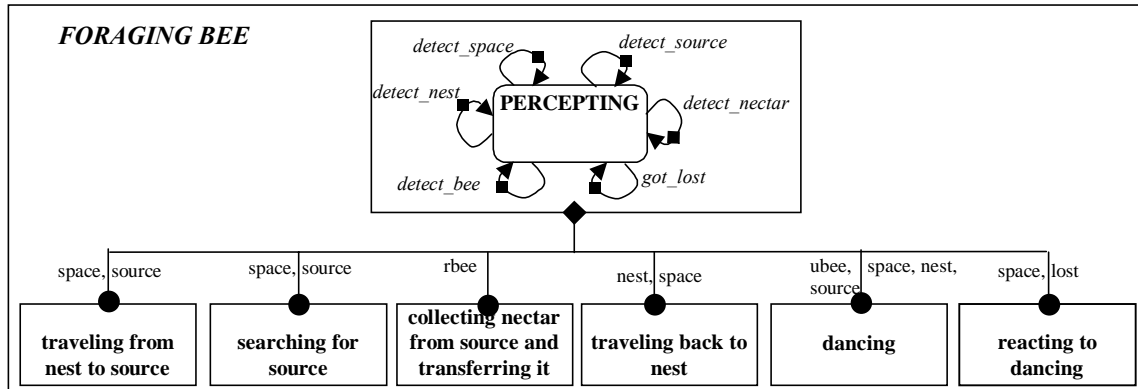


Fig. 8. An example of two communicating behaviours; an employed bee sends information about the source position to an unemployed bee.

The separate behaviours can be put together in a communicating X-machine model. Fig.9 shows the complete foraging bee system, which is made up of component X-machines, which communicate via channels. Each machine works separately and concurrently in an asynchronous manner. Each machine can read inputs from a communication channel instead of its standard input tape. Also, each machine can send a message through a communication channel that will act as input to functions of another component. The figure shows an extra component, i.e. the perception system of the bee, which provides percepts to various behaviours.

In addition, more machines can be modelled, as for example, the X-machine that builds an environment map (positions of obstacles, nest, food items etc.). Information held in the memory of this machine could be used to efficiently move around the environment, or even to model a pro-active behaviour for the agent. Thus, modelling of an agent can be incremental by providing components, which will advance further the level of intelligent behaviour.



9. Communicating X-machine modelling agent bee through aggregation of behaviours

The whole system works as follows: an employed bee accepts inputs from the environment, which cause transitions in the X-machine components that model its individual behaviours. Having found a source and after collecting nectar (appropriate transitions have been performed in source and environment X-machines), the bee returns to hive and on the sight of another foraging bee performs the dancing which, as shown earlier, transmits the information of the source position. An unemployed bee accepts the input from its communication port and changes its status to employed. It can then perceive inputs from the environment and travels to the source. The whole process may then be repeated. In parallel, other bees can do the same in an asynchronous manner.

The approach is practical, in the sense that the developer can separately model the components of an agent and then describe the way in which these components-behaviours communicate. Also, components can be re-used in other systems, since the only thing that needs to be changed is the communication part. For example, the behaviour for avoiding obstacles is a component of any biology-inspired agent, e.g. foraging bees or ants. The major advantage is that the methodology also lends itself to modular model checking and testing strategies in which X-machines are individually tested as components while communication is tested separately with existing methodologies, mentioned earlier.

MULTI-AGENT MODELS

Modelling multi-agent systems requires the consideration of the means of communicating between agents, in order to coordinate tasks, cooperate etc. Also, modelling of artificial environments in which agents act imposes the need of exchanging “messages” between agents and the environment. For example, a number of ants modelled as X-machines need to interact with their environment, which contains few seeds (food items) that are also modeled as X-machines. These two ants, which may be instances of the same model class, can communicate with the environment in order to achieve the desired behaviour, i.e. to lift a heavy seed that is far from the abilities of a single agent (Fig.10). Several behaviours are omitted for the sake of exposition. The ant is capable of lifting a food item only if the strength it possesses is bigger than the weight of a food item. In any other case, cooperation between ants is necessary, which can be achieved by communication of ants and the food item machine. The method

used in the previous section to describe communicating X-machines can also serve this purpose.

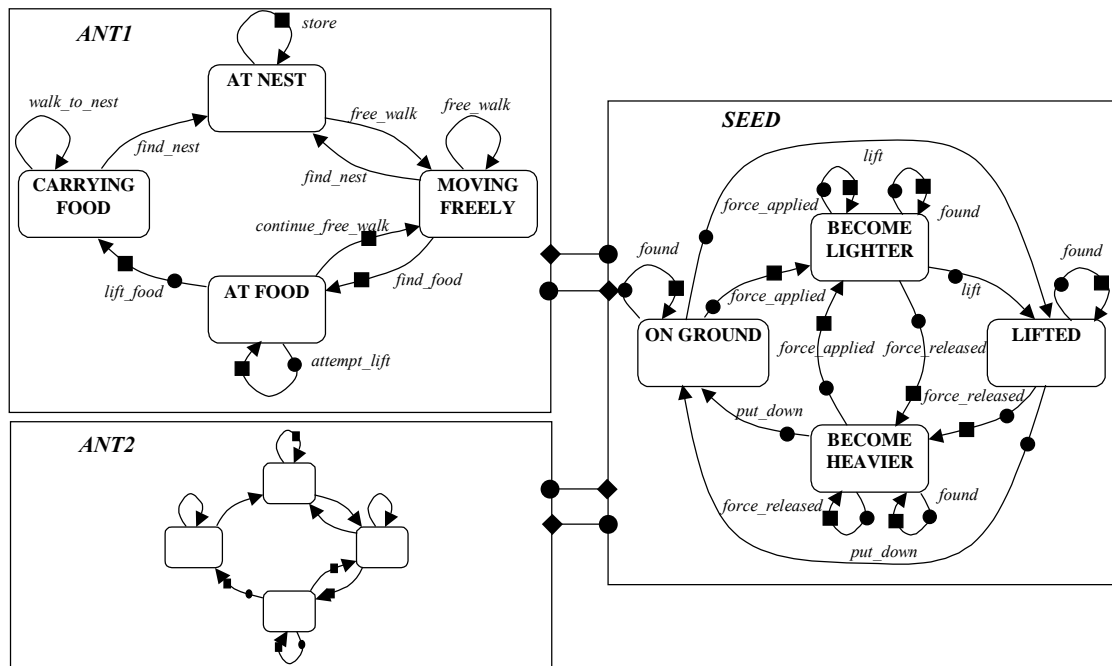


Fig. 10. Ants model cooperating in the lifting task through communication with the environment

In addition, one may require agents which resemble one another, i.e. they have a common set of behaviours, but extra individual behaviours which determine some task that characterise their individuality. For example, in a colony of foraging bees, some bees are responsible for collecting nectar from a source as well as have the ability to “inform” others about the location of the source (the dance of the foraging bee) while other bees are responsible for storing the nectar into the hive (Seely & Buhrman, 1999). Nevertheless, all of them have the ability to fly, receive nectar etc. Such situations can be modelled with X-machines, as long as there is a way to define classes of models and instances of these classes, which can inherit generic behaviours from the chain of hierarchy. Fig.11 demonstrates the whole multi-agent system that models the colony of the honey bees as well as the environment and its various components, such as the source and the nest.

The same happens when coordination is achieved by some other agent through scheduling and decomposition of a large task into smaller tasks, which are manageable by individual agents. Ready-made components may be used to complete the multi-agent system, as discussed before. If however, these components bear some incompatibility with the rest of the agents, communication and interaction protocols may be required. One can easily imagine X-machines that act as a synthetic glue between agents, modelling, for example, *KQML* parsers (Finin et al., 1997) or the *Contract Net protocol* (Davis & Smith, 1983).

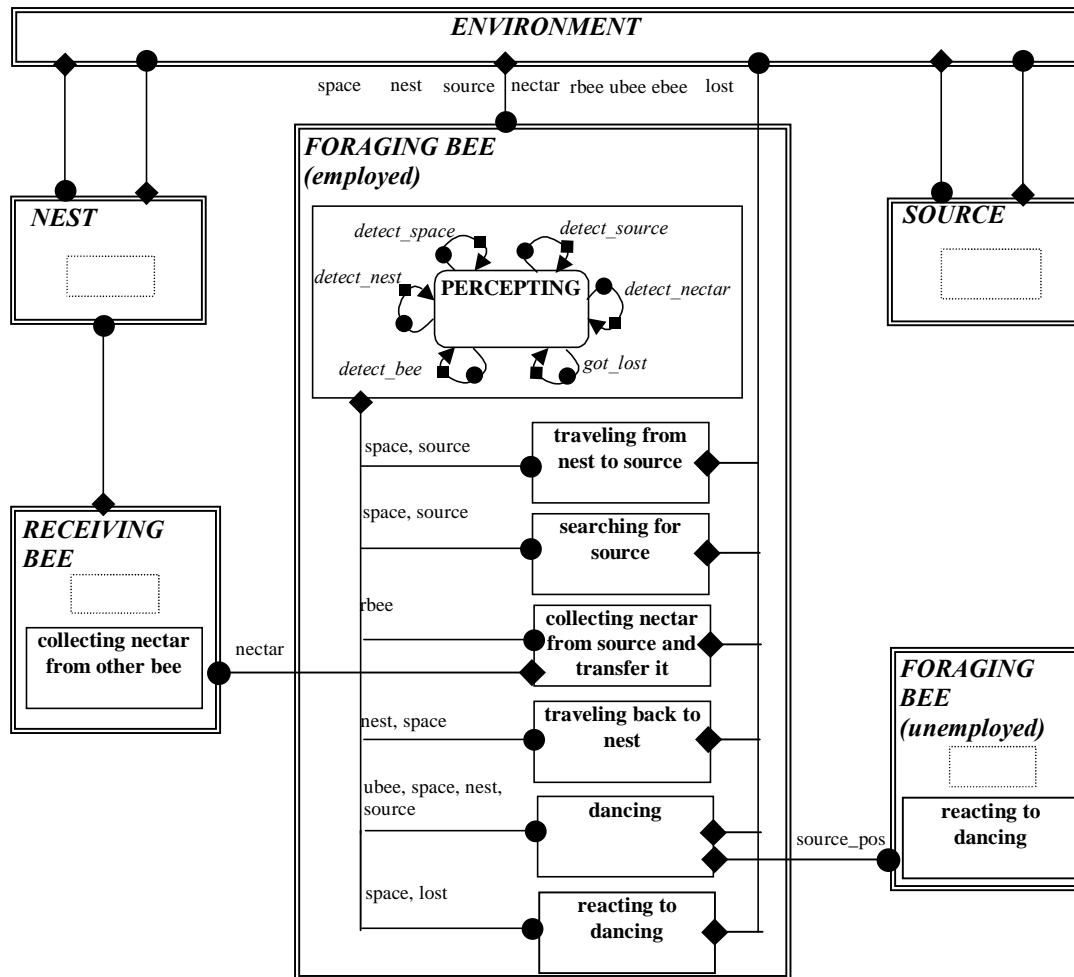


Fig. 11. The model of the honey bees multi-agent system and its interaction with the environment.

TOOLS

X-machines modelling is based on a mathematical notation, which, however, implies a certain degree of freedom, especially as far as definition 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 (Kapeti & Kefalas, 1999). Our aim was 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 (Fig.12). To avoid complex mathematical notation, the language symbols are completely defined in ASCII code. A model developed with XMDL consists of:

- the model for a component X-machine, and
- the coding referring to possible communication of this component with other X-machines.

Briefly, a XMDL based model is a list of definitions corresponding to the 8-tuple of the X-machine. The language also provides syntax for:

- use of built-in types such as integers, Booleans, sets, sequences, bags, etc.
- use of operations on these types, such as arithmetic, Boolean, set operations etc.
- definition of new types,
- definition of functions and the conditions under which they are applicable.

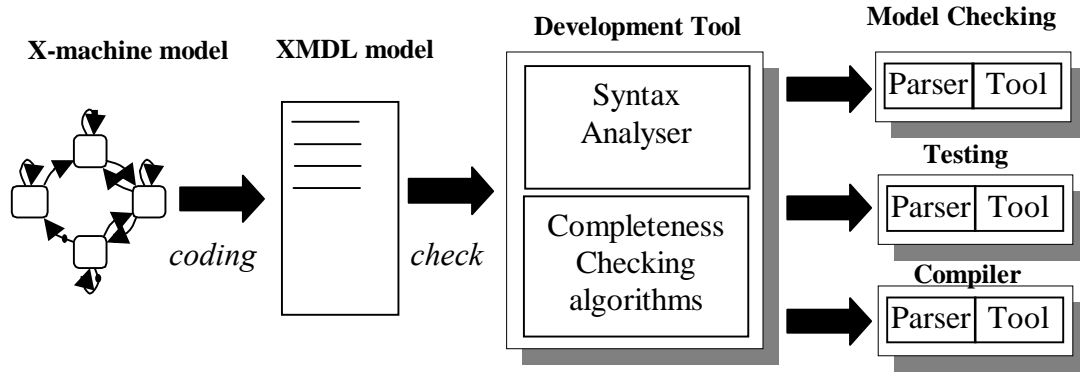


Fig. 12. The use of XMDL in system development.

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 “?”. The informative where in combination with the operator “<-“ is used to describe operations on memory values. The full syntax and semantics of XMDL can be found in (Kefalas, 2000). For example, the following list presents part of the XMDL code for the agent model described earlier (Fig.3):

```

#model ant.
#type coord = {-10 ... 10}.
#type position = (coord,coord).
#basic_types = [FOOD].
...
#input = {space, nest} union FOOD.
...
#memory (carrying, ant_position, food_positions).
#init_memory (none, (0,0), nil).
#init_state {at_nest}.

#states = {at_nest, at_food, moving_freely, going_back_to_nest, looking_for_food}.

#fun lift_food( (?f,?x,?y),(none,(?x,?y),?foodlist) )=
  if ?f belongs FOOD and (?x,?y) not_belongs ?foodlist then
    ("lifting food"),(?f,(?x,?y),<( ?x,?y) :: ?foodlist>)).
#fun find_food( (?f,?fpx,?fpy), (?food,(?x,?y),?foodlist) )=
  if ?f belongs FOOD and ?f not belongs ?foodlist then
    ("more food"),(?food,(?x,?y),<( ?fpx,?fpy) :: ?foodlist>)).
#fun drop_food( (nest,0,0), (?food,(?x,?y),?foodlist) )=
  ("dropping food"),(none,(0,0),?foodlist)).
#fun find_nest( (nest,0,0), (none,(?x,?y),?foodlist) )=
  ("found nest again"),(none,(0,0),?foodlist)).
...
#transition (at_nest, ignore_food) = at_nest.
#transition (at_nest, move) = moving_freely.
#transition (moving_freely, lift_food) = at_food.
...
#end.

```

In order to incorporate the semantics of communicating X-machines, the syntax of XMDL provides the following annotation:

```
#model <instance name> instance_of <model name>
  [with:
    #init_state = <instance initial state>.
    #init_memory <instance initial memory tuple>].
#communication of <model name>:
  <function name> reads from <model name>
  <function name> writes <message> to <model name>
  [where <expression> from (memory|input|output) <tuple>].
```

For example, the following list represents the communication part of an ant agent:

```
#model ant1 instance_of ant.
#communication of ant1:
  find_food reads from ant2.
#model ant2 instance_of ant.
#communication of ant2:
  lift_food writes (?f,?x,?y) to ant1
  where (?f,?x,?y) from input (?f,?y,?y).
```

The XMDL language is:

- orthogonal, since there are rather few primitive structures that can be combined in rather few ways,
- mark-up since it supplies the users with the freedom of a non-positioning language,
- strongly typed since it performs type checking, set checking and function checking, as well as exception handling, and
- very close to the mathematical notation.

DISCUSSION

We have demonstrated how formal methods and specifically X-machines can be used as the core method in intelligent agent engineering process. Firstly, we argued that X-machines are better suited to the requirements of agent systems than simple automata as far as their modelling is concerned. The use of memory structure and functions provide a better mapping to the requirements of an agent model and they are proved essential when simple automata fail to provide a finite model. Secondly, the relation of X-machines to *FSM* gives the opportunity to exploit existing techniques for both verification and testing. Verification is achieved through a variant of temporal logic adopted for X-machines, namely *XmCTL*, which can be utilised for model checking of agents, i.e. to verify whether certain desired properties hold in the agent model. Testing refers to the agent implementation and it is complete, in the sense that a variant of the W-method can automatically generate all possible test cases for the agent. Thirdly, a methodology for building complex agent systems through aggregation of behaviours as well as multi-agent systems is discussed. Communicating X-machines are demonstrated as a powerful extension to X-machines that facilitate modelling of large-scale systems, without the loss of the ability to use model checking and testing in each individual component.

Finally, a notation that makes X-machine practical was presented. The X-Machine Definition Language provides the appropriate syntax and semantics in order to make the formal method:

- practical, in the sense that complex mathematical definitions can be written using simple text,
- general, since the notation becomes standard and therefore the same model can be used throughout a variety of tools,
- modular, since it can accommodate component-based communicating X-machine model development, and
- disciplined, since modelling of components and communication between them are regarded as two separate developing activities.

Currently XMDL forms the basis for the development of various tools, such as automatic translations to programming languages such as *PROLOG* and *JAVA* (Kefalas, 2000) or transformations to other formal notations such as *Z* (Kefalas & Sotiriadou, 2000). Also, various other tools such as animators, test-case generators and model checkers have XMDL as an interchange language. Finally, XMDL is recently extended to accommodate certain object-oriented features of X-machine modelling as well as cardinality of communication in a multi-agent system (Kefalas et al., 2001).

CONCLUSION

Further work is required in both developing the tools further but also in building more examples of agent systems and carrying out the testing and verification described. Because the X-machine method is fully grounded in the theory of computation it is fully general and will be applicable to any type of computational task. The paradigm of the X-machine is also very convenient when it comes to implementing the models in an imperative programming language. In fact the translation is more or less automatic. The existence of the powerful testing method described lays the foundation for the method to be used in potentially critical applications. Finally the model checking developments will lead to a situation where one of the key issues in agent software engineering, namely how can we guarantee that the agent system constructed will exhibit the desired emergent behaviour can be solved, or at least substantial progress towards this goal will be achieved.

REFERENCES

- Attoui, A., & Hasbani, A. (1997). Reactive systems developing by formal specification transformations. Proceedings of the 8th International Workshop on Database and Expert Systems Applications (DEXA 97), (pp.339-344)
- Balanescu, T., Cowling, A.J., Gheorgescu, H., Gheorghe, M., Holcombe, M., & Vertan, C. (1999). Communicating stream X-machines systems are no more than X-machines. Journal of Universal Computer Science, 5 (9), 494-507
- Barnard, J. (1998). COMX: a design methodology using communicating X-machines. Journal of Information and Software Technology, 40, 271-280
- Benerecetti, M., Giunchiglia, F., & Serafini, L. (1999). A model checking algorithm for multiagent systems. In J. P. Muller, M. P. Singh, and A. S. Rao (Eds.). Intelligent Agents V (LNAI Volume 1555). (pp.163-176). Springer-Verlag
- Brazier, F., Dunin-Keplicz, B., Jennings, N., & Treur, J. (1995). Formal specification of multi-agent systems: a real-world case. Proceedings of International Conference on Multi-Agent Systems (ICMAS'95), (pp.25-32). MIT Press,

- Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics Automation*, 2 (7), 14-23
- Brooks, R. A. (1991). Intelligence without reason. In J. Mylopoulos, & R. Reiter (Eds.). (pp. 569-595). *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann
- Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., & Hwang, J. (1992). Symbolic model checking: 10^{20} states and beyond, *Information and Computation*, 98 (2), 142-170
- Chow, T.S. (1978). Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4 (3), 178-187
- Clarke, E., Wing, J. M. (1996). *Formal Methods: State of the Art and Future Directions*, *ACM Computing Surveys*, 28 (4), 626-643
- Clarke, E.M., Emerson, E.A., & Sistla, A.P. (1986). Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8 (2), 244-263
- Collinot, A., Drogul, A. & Benhamou, P. (1996). Agent oriented design of a soccer robot team. In *proceedings of the 2nd International Conference on Multi-Agent Systems*, 41-47
- Cowling, A.J., Gheorgescu, H., & Vertan, C. (2000). A structured way to use channels for communication in X-machines systems. *Formal Aspects of Computing*, 12, 485-500
- Davis, R., & Smith, R. (1983). Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20 (1), 63-109
- Deneubourg, J.-L., Aron, S., Goss, S., & Pasteels J.-M. (1990). The self-organizing exploratory pattern of the Argentine ant. *Journal of Insect Behavior*, 3, 159-68
- Dorigo, M., & Di Caro, G. (1999). The ant colony optimization meta-heuristic. In D. Corne, M. Dorigo, & F. Glover (Eds.), *New Ideas in Optimization*, (pp.11-32). McGraw-Hill,
- Eilenberg, S. (1974). *Automata, Machines and Languages*. Vol. A. Academic Press.
- Eleftherakis, G., & Kefalas P. (2001). Towards model checking of finite state machines extended with memory through refinement. In G. Antoniou, N. Mastorakis, & O. Panfilov (Eds.). (pp.321-326). *Advances in Signal Processing and Computer Technologies*, World Scientific and Engineering Society Press
- Eleftherakis, G., & Kefalas, P. (2001). Model checking safety critical systems specified as X-machines. *Matematica-Informatica, Analele Universitatii Bucharest*, 49 (1), 59-70
- Emerson, E.A., & Halpern, J.Y., (1986). Sometimes and not never revisited: On branching time versus linear time, *Journal of the ACM*, 33, 151-178
- Ferber, J. (1996). *Reactive distributed artificial intelligence: Principles and applications*. In *Foundations of distributed Artificial Intelligence*, (pp.287-314). John Wiley & Sons, Inc.
- Finin, T., Labrou, Y., & Mayfield, J. (1997). KQML as an agent communication language. In J.M Bradshaw (Ed.). *Software Agents*, (pp. 291-316). AAAI Press
- Fisher, M. & Wooldridge, M. (1997). On the Formal Specification and Verification of Multi-Agent Systems. *International Journal of Cooperating Information Systems*, 6(1), 37-65
- Futatsugi, K., Goguen, J., Jouannaud, J.-P., & Meseguer, J. (1985). Principles of OBJ2. In B. Reid (Ed.). *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, (pp.52-66). Association for Computing Machinery
- Georghe, M., Holcombe, M., & Kefalas, P. (2001). Computational models for collective foraging, *4th International Workshop on Information Processing in Cells and Tissues*, Lueven, Belgium

- Harel, D. (1987). Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8 (3)
- Holcombe, M. (1988). X-machines as a basis for dynamic system specification. *Software Engineering Journal*, 3 (2), 69-76
- Holcombe, M., & Ipaté, F. (1998). *Correct systems: Building a business process Solution*. Springer Verlag, London
- Inverno, d' M., Kinny, D., Luck, M., & Wooldridge, M. (1998). A formal specification of dMARS. In M.P.Singh, A.Rao, and M.J.Wooldridge (Eds.). *Intelligent Agents IV (LNAI Volume 1365)*, (pp.155-176). Springer-Verlag
- Ipaté, F., & Holcombe, M. (1998). Specification and testing using generalised machines: a presentation and a case study. *Software Testing, Verification and Reliability*, 8, 61-81
- Jennings, N.R. (2000). On agent-based software engineering. *Artificial Intelligence*, 117, 277-296
- Jennings, N.R. (2001). An agent-based approach for building complex software systems. *Communications of the ACM*, 44 (4), 35-41
- Jones, C. B. (1990). *Systematic software development using VDM (2nd ed.)*. Prentice-Hall
- Kapeti, E., & Kefalas, P. (2000). A design language and tool for X-machines specification. In D.I.Fotiadis, S.D.Nikolopoulos (Eds.). *Advances in Informatics*, (pp.134-145). World Scientific Publishing Company
- Kefalas, P. (2000). Automatic Translation from X-machines to Prolog. Technical Report TR-CS01/00, Dept. of Computer Science, CITY Liberal Studies
- Kefalas, P. (2000). XMDL user manual: version 1.6. Technical Report TR-CS07/00, Dept. of Computer Science, CITY Liberal Studies
- Kefalas, P., & Sotiriadou, A. (2000). A compiler that transforms X-machines specification to Z. Technical Report TR-CS06/00, Dept. of Computer Science, CITY Liberal Studies
- Kefalas, P., Eleftherakis, G., & Kehris, E. (2001). Modular modelling of large-scale systems using communicating X-machines, In Y.Manolopoulos and S.Evripidou (Eds.). *Proceedings of the 8th Panhellenic Conference in Informatics*, (pp.20-29), Livanis Publishing Company
- Kripke, S. (1963). A semantical analysis of modal logic I: normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen Mathematik*, 9, 67-96
- McMillan, K.L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers
- Odell, J., Parunak, H. V. D., & Bauer, B. (2000). Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*
- Rao, A.S., & Georgeff, M. (1995). BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, (pp.312-319)
- Rao, A.S., & Georgeff, M.P. (1993). A model-theoretic approach to the verification of situated reasoning systems. In R.Bajcsy (ed.). *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, (pp.318-324). Morgan Kaufmann
- Reisig, W. (1985). *Petri nets - an introduction*. EATCS Monographs on Theoretical Computer Science, 4, Springer-Verlag
- Rosenschein, S. R., & Kaelbling, L. P. (1995). A situated view of representation and control. *Artificial Intelligence* 73 (1-2) pp 149-173,

Seeley, T.D., and Buhrman, S.C. (1991). Group decision making in swarms of honey bees. *Behavioral Ecology and Sociobiology*, 45, 19-31

Spivey, M. (1989). *The Z notation: A reference manual*. Prentice-Hall

Vries, H. de, Biesmeijer, J.C. (1998). Modelling collective foraging by means of individual behaviour rules in honey-bees. *Behavioral Ecology and Sociobiology*, 44 109-124.

Wooldridge, M., & Ciancarini, P. (2001). Agent-oriented software engineering: The state of the art. To appear in the *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Co.

Wulf, W. A., Shaw, M., Hilfinger, P.N., & Flon, L. (1981). *Fundamental structures of computer science*. Addison-Wesley,

Young, W. D. (1991). Formal Methods versus Software Engineering: Is There a Conflict? In *Proceedings of the Fourth Testing, Analysis, and Verification Symposium*, (pp. 188-899)