Formal Verification of Agent Models

George Eleftherakis, Petros Kefalas, Anna Sotiriadou
Computer Science Department
City Liberal Studies
Affiliated College of the University of Sheffield
13 Tsimiski Str., 54624 Thessaloniki, Greece
{eleftherakis, kefalas, sotiriadou}@city.academic.gr

Abstract

Intelligent agent engineering has proved to be a difficult task due to the inherent complexity of agent systems. Although many practical implementations of agent systems exist, their development was hardly based on methods that can prove their validity. Model checking is a formal verification technique which determines whether certain properties are satisfied by a system model. Prerequisites for applying model checking are (i) to view the agent as a Finite State Machine, and (ii) to be able to express the properties in a powerful specification formalism, such as Temporal Logic. Finite State Machines are too simple to capture the modelling needs of agents that normally require non-trivial data structures as well as complex control over these data structures and over the states in which agents can exist. X-Machines is a formal method that satisfies these requirements by introducing memory structure into finite state machines as well as computable functions between states of a model. With existing logics it is obscure how one can describe properties that refer to the memory data structure of an agent model. Thus, a new approach to model checking of agent systems should be introduced. This paper defines the syntax and the semantics of a new logic, namely XmCTL, which extends temporal logic with memory quantifiers, thus facilitating model checking targeted to agent models. The use of XmCTL and the importance of verification in agent system development are demonstrated through a simple example.

Keywords: Agents, Formal Verification, Temporal Logic

1 Introduction

An agent is an encapsulated computer system that is situated in some environment and is capable of flexible, autonomous action in that environment in order to meet its design objectives [Jen00]. Although agent-oriented software engineering aims to manage the inherent complexity of software systems [WoCi01], there is still no evidence to suggest that any methodology proposed leads towards "correct" systems. In agent oriented engineering, there have been several attempts to use formal methods, each one focusing on different aspects of agent systems development [AtHa97, BDJT95]. So far, little attention has been paid in formal methods that could facilitate all crucial stages of "correct" system development; modelling, verification and testing.

X-machines is a formal method which closely suits the needs of agent development, offering an intuitive and practical way of modelling [HoIp98] and at the same time a formal testing strategy to test the implementation against the X-machine model [IpHo97].

Having designed a model for an agent and before starting implementing this model, 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 is based on the exhaustive exploration of a given state space trying to determine whether a given property is satisfied by a system. 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 early 80s two different teams by Quielle and Sifakis [QuSi81] and by Clarke and Emerson [ClEm81], working in parallel, proposed temporal logic model checking algorithms. In *Temporal Logic Model Checking* [CES86] a property is expressed as a formula in a certain temporal logic, usually the *Computation Tree Logic* (CTL). In this context the most usual definition of a model is as a labelled state transition graph, also called *Kripke* structure $K = \langle Q, R, L \rangle$ [Kri63] where: i) Q is a non-empty set of states, ii) R is a binary relation on Q, i.e. $R \subseteq Q \times Q$, which shows which states are related to other states, iii) $L : Q \to 2^{Prop}$ is a truth assignment function that shows which propositions are true in each state, where $Prop$ is the set of atomic propositions. The semantics of CTL are defined with respect to a Kripke structure K [EmHa86].

Temporal Logic (TL) is an extension of logic through operators that handle the notion of time. Using these operators TL gives the ability to argue about "when" logical expressions are true. In CTL [CES86] each of the temporal operators must be preceded by either **A** (for all paths) or **E** (there exists path) path quantifiers. The five basic temporal operators of CTL are:

- **X** (next time) requires that a property holds at the following state,
- **F** (in the Future, eventually) a property will hold at some state on a path,
- **G** (Globally, always) a property holds at every state on a path,
- **U** (Until) combines two properties, and p **U** q holds in the model if there is a state in a path that the second property (q) holds and also the first property (p) holds in every preceding state on the path, thus p holds in a path until q holds,

- **R** (Release) also requires two properties and is the dual operator of **U** . $p\,\mathbf{R}\,q$ holds in the model if the second property (q) holds along a path up to and including the initial state where the first property (p) holds, however without requiring p to hold eventually.

CTL facilitates model checking in formal models that resemble a Kripke structure, like the Finite State Machines (FSM). However, CTL is not expressive enough to facilitate model checking of FSM extended with memory.

In this paper, we argue that there is a need for appropriate extensions of CTL so that temporal logic formulae are able to describe specifications suitable to refer to the memory structure of the model. A X-machine is a general computational machine that is like a FSM extended with a memory structure which models the data of a system. Unlike FSM, in a X-machine transitions are not labeled with simple inputs but with functions that operate on inputs and memory, allowing the machine to be more expressive and flexible than the FSM and able to model both the control and the data part of a system [HoIp98]. Thus we will introduce an extension of temporal logic that will facilitate the verification of X-machine models. With the use of an agent example we will demonstrate the feasibility of verifying agent models expressed as X-machines.

2 Modelling agents as FSM extended with memory

Many biological processes seem to behave like agents, e.g. a colony of ants. Complex optimisation problems have been solved based on such behaviour [DoDi99]. For example an ant has the important task to find food and carry it to its nest. This can be accomplished by searching for food at random or by following pheromone trails. Once food is found the ant should leave a pheromone trail while travelling back to its nest, thus communicating to other ants the destination of a source where food may be found. When the nest is found the ant drops the food. While moving the ant should avoid obstacles. Thus the ant receives input from the environment and acts upon these inputs according to the "state" in which the ant is in. Clearly this is the behaviour of a reactive agent. Such reactive agents can be fairly easily modelled by a FSM in a rather straightforward way by specifying the states and the inputs to be used for state transitions.

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 [Bro86, RoKa95] 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 reactiveness [GHK01].

X-machine is a formal method introduced by Eilenberg [Eil74], which is capable of modelling both the data and the control of a system. X-machines employ a diagrammatic approach of modelling the control by extending the expressive power of FSM.

Transitions between states are no longer performed through simple input symbols but through the application of functions. In contrast to FSM, X-machines are capable of modelling non-trivial data structures by employing a memory, which is attached to the X-machine. Functions receive input symbols and memory values, and produce output while modifying the memory values. Holcombe proposed X-machines as a basis for a possible specification language [Hol88]. Stream X-machines are defined as X-machines with input and output sets of streams of symbols [HoIp98].

A *stream X-machine* is an 8-tuple $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ 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 $\mathcal{M}$, a finite set of partial functions ϕ that map an input and a memory state to an output and a new memory state, $\phi : \Sigma \times M \to \Gamma \times M$
- F is the next state partial function that given a state and a function from the type Φ, denotes the next state. F is often described as a transition state diagram. $F : Q \times \Phi \to Q$
- q_0 and m_0 are the initial state and memory respectively.

The formal model, as a X-machine, of an ant that searches for food, but also remembers food positions in order to set up its next goals is defined in table 1. The behaviour of obstacle avoidance is omitted for simplicity. Fig. 1 shows the state transition diagram, where the transitions are labeled with functions.

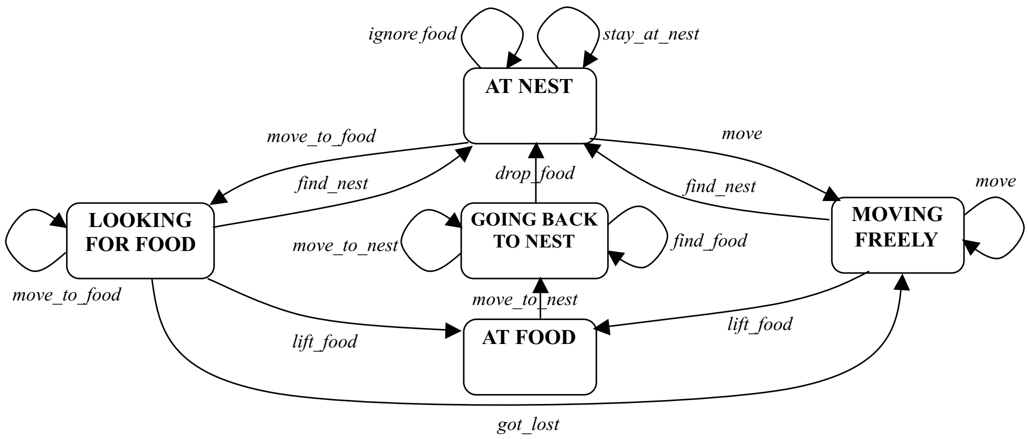

Figure 1: An ant X-machine model

Table 1: Formal model of an ant

The *X-machine* is defined as an 8-tuple $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ where:	
Input set Σ	$\Sigma = (\{space, nest\} \cup FOOD) \times COORD \times COORD$
Output set Γ	$\Gamma = \{$ *“moving freely”, “moving to nest”, “dropping food”*, $\ldots\}$
Set of states Q	$Q = \{$ *At Nest, Moving Freely, At Food, Going Back To Nest, Looking for Food*$\}$
Memory M	$M = (FOOD \cup \{none\}) \times (COORD \times COORD) \times seq(COORD \times COORD)$
Initial state q_0	$q_0 =$ “At Nest”
Initial memory m_0	$m_0 =$ *(none, (0,0), nil)*
Next state function F	$F : Q \times \Phi \rightarrow Q$ shown diagrammatically in figure 1
Type Φ of the machine is the set of functions $\phi : \Sigma \times M \rightarrow \Gamma \times M$ The functions are defined next using the notation: $\phi(\sigma, m) = (\gamma, m')$ *if condition*	move*((space,xs,ys), (none, (x,y), nil))* $\rightarrow$ *(“moving freely”, (none,(xs,ys),nil))* *if next(x,y,xs,ys)* move-to-food*((space,xs,ys), (none, (x,y), <(fpx,fpy)::rest>))* $\rightarrow$ *(“moving to food”, (none,(nx,ny), <(fpx,fpy)::rest>))* *if next(x,y,xs,ys)* $\wedge$ *closer-to-food(fpx,fpy,xs,ys)* move-to-nest*((space,xs,ys), (food, (x,y), foodlist))* $\rightarrow$ *(“moving to nest”, (food,(nx,ny),foodlist))* *if food* $\in$ *FOOD* $\wedge$ *next(x,y,xs,ys)* $\wedge$ *closer-to-nest(xs,ys)* lift-food*((f,x,y), (none, (x,y), foodlist))* $\rightarrow$ *(“lifting food”, (f, (x,y), <(x,y)::foodlist>))* *if f* $\in$ *FOOD* $\wedge$ *(x,y)* $\notin$ *foodlist* lift-food*((f,x,y), (none, (x,y), foodlist))* $\rightarrow$ *(“lifting food”, (f, (x,y), foodlist))* *if f* $\in$ *FOOD* $\wedge$ *(x,y)* $\in$ *foodlist* find-food*((f,fpx,fpy), (food, (x,y), foodlist))* $\rightarrow$ *(“more food”, (food, (x,y), <(fpx,fpy)::foodlist>))* *if f* $\in$ *FOOD* $\wedge$ *f* $\notin$ *foodlist* drop-food*((nest,0,0), (food, (x,y), foodlist))* $\rightarrow$ *(“droping food”, (none, (0,0), foodlist))* find-nest*((nest,0,0), (none, (x,y), foodlist))* $\rightarrow$ *(“found nest again”, (none, (0,0), foodlist))* got-lost*((space,fpx,fpy), (none, (x,y), <(fpx,fpy)>))* $\rightarrow$ *(“got lost”, (none, (x,y), nil))* *if next(x,y,xs,ys)* got-lost*((space,fpx,fpy), (none, (x,y), <(fpx,fpy)>))* $\rightarrow$ *(“got lost”, (none, (x,y), nil))* *if f* $\in$ *FOOD*

where the functions next, closer-to-nest and closer-to-food are considered as external functions, i.e. functions defined elsewhere (possibly as X-machines themselves):

next: COORD $\times$ COORD $\times$ COORD $\times$ COORD $\rightarrow$ BOOLEAN

closer-to-nest: COORD $\times$ COORD $\rightarrow$ BOOLEAN

closer-to-food: COORD $\times$ COORD $\times$ COORD $\times$ COORD $\rightarrow$ BOOLEAN

3 Model checking agent models

Having defined the formal model of the agent it would be desirable to have a mechanism to check if the proposed model has all the desired properties. For example in the formal model of the ant presented in section 2 we would like to prove that whenever the ant will pickup food it will eventually drop the food. Having the formal model, there has to be a way to express properties like the one stated before in a mathematical language in order to enable the use of an automated and formal verification. Model checking demonstrated to be a powerful, automated formal verification technique. Temporal logic model checking is using mostly CTL which is a variation of temporal logic appropriate to express properties on models expressed formally as Kripke structures.

In X-machines, the search of some properties P of the model being true or false cannot be applied in a straightforward manner, since these properties are implicitly expressed in the X-machine memory values. Thus, checking whether a property $p \in P$ is valid in some states of the X-machine means whether there are some states in which some memory values satisfy the property p. For example, in order to verify that a memory variable will never exceed a maximum value in any of the states, a model checker needs to search through all possible states as well as all possible instances of memory. Therefore, the appropriate model that facilitates model checking $\langle Q, R, L\rangle$ should include: i) Q is the set of all possible states of the X-machine combined with all possible instances of memory in each state, ii) R is the set of transitions between states in Q, iii) L is the truth assignment function, i.e. given a member in Q shows which properties are true depending on the values of this memory instance.

Bearing the above, model checking of a X-machine model for specific properties can be achieved through the transformation of the X-machine into the form $\langle Q, R, L\rangle$. The resulting state space (Q, R) resembles a FSM $(\Sigma, \Gamma, Q_{fsm}, T, q_{0fsm})$ where: i) Σ is a finite set that is called the input alphabet. ii) Γ is a finite set that is called the output alphabet. iii) Q_{fsm} is the finite set of states. iv) T is the (partial) transition function, $T : Q_{fsm} \times \Sigma \to Q_{fsm} \times \Gamma$, ($T$ is a labelled R). v) q_{0fsm} is an initial state, that encapsulates memory values which correspond to properties in each of its states. Thus with the use of CTL formulas to express a desired specification the above Kripke structure could be verified [ElKe01]. But although this process demonstrates the feasibility of model checking X-machine models, it creates two problems; i) the loss of expressiveness that the X-machine possesses, e.g., CTL is not expressive enough to describe that a properly p holds in some but not all memory instances of all states of the X-machine, and ii) the combinatorial explosion.

In order to overcome the lack of expressiveness, there is a need for an appropriate formalism for expressing properties of state transition systems extended with memory. The proposed logic (XmCTL) is an extension of CTL that will facilitate effective model checking of X-machines through the use of operators that quantify memory instances within a single state. Thus having the X-machine model of the agent it will be easier and more intuitive to write the formula that expresses the desired property in XmCTL.

4 XmCTL

4.1 Definitions

Definition 4.1 *$MProp$ is the set of all predicates composed of instances of memory variables and/or atoms.*

Definition 4.2 *A state q of the X-machine $\mathcal{M}$ is called x-state.*

Definition 4.3 *The set of all memory instances of all x-states q is denoted as Q_{fsm} and it is the set of states of the equivalent finite state machine derived from the exhaustive refinement of the X-machine $\mathcal{M}$. A memory instance of a x-state is denoted as qm ($qm \in Q_{fsm}$). The notation qm^i is also used to denote the i-th memory instance of the q x-state.*

Definition 4.4 *$\tau : Q_{fsm} \rightarrow 2^{MProp}$ is the truth assignment function that given a x-state memory instance qm^i, returns a set of all the propositions $p \in MProp$ that are true in the specific x-state q for the specific i-th memory instance.*

Definition 4.5 *A x-path π in a X-machine $\mathcal{M}$ is defined as an infinite sequence of states of $\mathcal{M}$, $\pi = q_0, q_1, \ldots$ such that for every $i \geq 0, \exists \phi \in \Phi : (q_i, \phi, q_{i+1}) \in F$ (an infinite branch in the corresponding to $\mathcal{M}$ computation tree). Also the notation π^i is used to denote the suffix of π starting at state q_i.*

4.2 Syntax

In this section the syntax of XmCTL formulas is defined. The temporal operators used in XmCTL are the operators of CTL described in section 1 with the addition of two new memory quantifiers: $\mathbf{M_x}$ and $\mathbf{m_x}$. $\mathbf{M_x}$ (for all memory instances) requires that a property holds at all possible memory instances of a x-state. $\mathbf{m_x}$ (there exists memory instance) requires that a property holds at some memory instances of a x-state. A class of x-state memory formulas (XSM), x-state formulas (XS) and x-path formulas (XP) are defined inductively:

XSM1. if $p \in MProp$, then p is a *x-state memory formula*,

XSM2. if a and b are x-state memory formulas, then $\neg a$, $a \vee b$, $a \wedge b$, are *x-state memory formulas*,

XS1. if a is x-state memory formula, then $\mathbf{M_x}a$ and $\mathbf{m_x}a$ are *x-state formulas*,

XS2. if f_1 and f_2 are x-state formulas, then $\neg f_1$, $f_1 \vee f_2$, $f_1 \wedge f_2$, are *x-state formulas*,

XP1. if f_1 and f_2 are x-state formulas, then $\mathbf{X}f_1$, $\mathbf{F}f_1$, $\mathbf{G}f_1$, f_1 $\mathbf{U}$ f_2, and f_1 $\mathbf{R}$ f_2 are *x-path formulas.*

XS3. if g is a x-path formula, then $\mathbf{A}g$, $\mathbf{E}g$ are *x-state formulas*,

A valid XmCTL formula is any x-state formula.

4.3 Semantics

In CTL, the semantics are defined with respect to a Kripke structure. Here, the semantics of XmCTL will be defined with respect to the X-machine model.

Definition 4.6 *The notation $\mathcal{M}, q \models f$, means that f (which is a x-state formula) holds at x-state q in the model $\mathcal{M}$. If p is a x-state memory formula, then the notation $\mathcal{M}, qm^i \models p$ means that p holds in the x-state memory instance qm^i in the model $\mathcal{M}$. In the case that g is a path formula, then the notation $\mathcal{M}, \pi \models g$ means that g holds along path π in the model $\mathcal{M}$. Usually if the X-machine structure $\mathcal{M}$ is clear from the context, it is omitted. Assuming that a, b are x-state memory formulas, f, f_1 and f_2 are x-state formulas, and g_1 and g_2 are x-path formulas, the relation $\models$ is defined inductively below:*

XSM1. $\mathcal{M}, qm^i \models p \Leftrightarrow p \in \tau(qm^i)$, where $p \in MProp$

XSM2. $\mathcal{M}, qm^i \models \neg a \Leftrightarrow \mathcal{M}, qm^i \not\models a$,

$\mathcal{M}, qm^i \models a \wedge b \Leftrightarrow \mathcal{M}, qm^i \models a$ and $\mathcal{M}, qm^i \models b$,

$\mathcal{M}, qm^i \models a \vee b \Leftrightarrow \mathcal{M}, qm^i \models a$ or $\mathcal{M}, qm^i \models b$,

XS1. $\mathcal{M}, q \models \mathbf{M_x} a \Leftrightarrow$ for all $i \geq 0$, $\mathcal{M}, qm^i \models a$,

$\mathcal{M}, q \models \mathbf{m_x} a \Leftrightarrow$ there exists $i \geq 0$, $\mathcal{M}, qm^i \models a$,

XS2. $\mathcal{M}, q \models \neg f \Leftrightarrow \mathcal{M}, q \not\models f$,

$\mathcal{M}, q \models f_1 \wedge f_2 \Leftrightarrow \mathcal{M}, q \models f_1$ and $\mathcal{M}, q \models f_2$,

$\mathcal{M}, q \models f_1 \vee f_2 \Leftrightarrow \mathcal{M}, q \models f_1$ or $\mathcal{M}, q \models f_2$,

XP1. $\mathcal{M}, \pi \models \mathbf{X} f \Leftrightarrow \mathcal{M}, first(\pi^1) \models f$,

$\mathcal{M}, \pi \models \mathbf{F} f \Leftrightarrow \mathcal{M}, first(\pi^i) \models f$, there exists $i \geq 0$,

$\mathcal{M}, \pi \models \mathbf{G} f \Leftrightarrow \mathcal{M}, first(\pi^i) \models f$, for all $i \geq 0$,

$\mathcal{M}, \pi \models f_1 \mathbf{U} f_2 \Leftrightarrow$ there exists $k \geq 0$: $\mathcal{M}, first(\pi^k) \models f_2$ and for all $0 < j < k$ $\mathcal{M}, first(\pi^j) \models f_1$,

$\mathcal{M}, \pi \models f_1 \mathbf{R} f_2 \Leftrightarrow$ for all $j \geq 0$, if for every $i \leq j$ $\mathcal{M}, first(\pi^i) \not\models f_1$ then $\mathcal{M}, first(\pi^j) \models f_2$,

XS3. $\mathcal{M}, q \models \mathbf{A} g \Leftrightarrow \mathcal{M}, \pi \models g$ for all x-paths π with $first(\pi) = q$,

$\mathcal{M}, q \models \mathbf{E} g \Leftrightarrow$ there exists x-path π with $first(\pi) = q$ such that $\mathcal{M}, \pi \models g$.

4.4 Using XmCTL

The resulting logic XmCTL can verify agent models expressed as X-machine against the requirements, since it can prove that certain properties, which implicitly reside on the memory of X-machine are true [ElKe01]. For example, in an agent whose task is to carry food to its nest as in example of Fig. 1, model checking can verify whether eventually food will be dropped in the nest by the formula:

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

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

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

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:

$$\neg\mathbf{EFm_x}[(m_1 \neq \mathit{none}) \wedge (m_3 = \mathit{nil})]$$

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

5 Discussion and conclusions

The syntax and semantics of XmCTL are defined in order to improve expressiveness with respect to agent systems which are modelled as X-machines. However, efficiency in model checking a X-machine is still an issue under consideration due to the state explosion if exhaustive refinement to a “flat” FSM is applied. Efficiency may be substantially improved depending on the number of properties involved. For example, if the properties in the XmCTL formula refer to the whole memory tuple then the X-machine should be exhaustively refined into a FSM. This is inevitable, but it would have happened anyway if the system was modelled as a FSM from scratch. But if some of the elements in the memory tuple do not correspond to a given property in a XmCTL formula then the memory values of this element should not participate in the refinement of the X-machine. Thus, selective refinement can be performed [ElKe01] which will reduce the number of states in it and therefore the refined X-machine will contain exactly the necessary states for the model checking process.

X-machines are able to model both the control and the data part of a system and therefore it possesses valuable characteristics that are desirable to software engineering of agent systems. A framework for formal development of systems proposed in [Ele01] uses X-machines as a formal modelling language [KEK00], a testing strategy to check the implementation against the X-machine model [IpHo97] and a verification technique to prove the validity of the model [ElKe00]. By applying this framework to agent systems it is possible to assure that all "desired" properties of an agent hold in the final product. The proposed logic has been specifically designed for the X-machine formalism and demonstrates the feasibility of verification in agent models. Having set up the theoretical framework, future work needs to be done on implememnting a model checker, which provided an agent model and XmCTL formulas will prove the validity of this model.

References

[AtHa97] Attoui A. and Hasbani A., "Reactive systems developing by formal specification transformations", In Proceedings of the 8th International Workshop on Database and Expert Systems Applications (DEXA 97), 1997, pp. 339-344.

[BDJT95] Brazier F., Dunin-Keplicz B., Jennings N., Treur J., "Formal specification of multi-agent systems: a real-world case", In Proceedings of International Conference on Multi-Agent Systems (ICMAS'95), MIT Press, 1995, pp. 25-32.

[Bro86] Brooks R. A., "A robust layered control system for a mobile robot", *IEEE Journal of Robotics Automation*, Vol. 2, No.7, 1986, pp. 14-23.

[CES86] Clarke E.M., Emerson E.A. and Sistla A.P., "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM Trans. Programming Languages and Systems*, Vol.8, No.2, Apr. 1986, pp. 244-263.

[ClEm81] Clarke E.M. and Emerson E.A., "Synthesis of synchronization skeletons for branching time temporal logic", In Proc. IBM Workshop on Logics of Programs, Lecture Notes in Computer Science, Vol.131, Springer-Verlag, 1981, pp.52-71.

[DoDi99] Dorigo M., and Di Caro G., "The ant colony optimization meta-heuristic", In D. Corne, M. Dorigo, and F. Glover (Eds.), New Ideas in Optimization, McGraw-Hill, 1999, pp.11-32.

[Eil74] Eilenberg S., Automata Machines and Languages, Vol. A, Academic Press, 1974.

[Ele01] Eleftherakis G., "A Formal Framework for Modelling and Validating Medical Systems", In MEDINFO 2001, V.Patel, R.Rogers and R.Haux (Eds.), IOS Press, Sep. 2001, pp.13-17.

[ElKe00] Eleftherakis G, and Kefalas P., "Model Checking Safety-Critical Systems Specified as X-machines", *An. Univ. Bucur. Mat. Inform.*, Vol. 49, No.1, 2000, pp. 59-70.

[ElKe01] Eleftherakis G, and Kefalas P., "Towards Model Checking of Finite State Machines Extended with Memory through Refinement", In G.Antoniou, N.Mastorakis, and O.Panfilov (Eds.), Advances in Signal Processing and Computer Technologies, World Scientific and Engineering Society Press, July 2001, pp.321-326.

[EmHa86] Emerson E.A. and Halpern J.Y., "'Sometimes' and 'not never' revisited: On branching time versus linear time", *Journal of ACM*, Vol.33, 1986, pp.151-178.

[GHK01] Gheorghe M., Holcombe M., Kefalas P., "Computational Models of Collective Foraging", 4th International Workshop on Information Processing in Cells and Tissues, IPCAT 2001, Luven, Belgium, August 2001.

[Hol88] Holcombe M., "X-machines as a basis for dynamic system specification", *Software Engineering Journal*, Vol.3, No.2, 1988, pp. 69-76.

[HoIp98] Holcombe M. and Ipate F., *Correct Systems: Building a Business Process Solution*, Springer Verlag, London, 1998.

[IpHo97] Ipate F. and Holcombe M., "An integration testing method that is proved to find all faults", *International Journal of Computer Mathematics*, Vol.63, No.3, 1997, pp. 159-178.

[Jen00] Jennings N.R., "On agent-based software engineering", *Artificial Intelligence*, Vol.117, No.2, 2000, pp. 277-296.

[KEK00] Kehris E., Eleftherakis G., and Kefalas P., "Using X-machines to Model and Test Discrete Event Simulation Programs", In Systems and Control: Theory and Applications, N. Mastorakis (ed.),, World Scientific and Engineering Society Press, July 2000, pp. 163-168.

[Kri63] Kripke S.A.,"Semantical analysis of modal logic I: normal propositional calculi", *Z. Math. Logik Grund. Math.*, Vol.9, 1963, pp.67-96.

[QuSi81] Quielle J.P. and Sifakis J., "Specification and verification of concurrent systems in CESAR", In Proc. 5th Int. Symp. on Programming, 1981, pp. 337-350.

[RoKa95] Rosenschein S. R., and Kaebling L. P., "A situated view of representation and control", *Artificial Intelligence*, Vol.73, No.1-2, 1995, pp. 149-173.

[WoCi01] Wooldridge M. and Ciancarini P., "Agent-Oriented Software Engineering: The State of the Art", In P. Ciancarini and M. Wooldridge (Eds.), Agent-Oriented Software Engineering, Lecture Notes in AI, Vol.1957, Springer-Verlag, Jan 2001, pp. 337-350.