Prevention of harmful behaviors within cognitive and autonomous agents

Caroline Chopinaud[1] and **Amal El Fallah Seghrouchni**[2] and **Patrick Taillibert**[3]

Abstract. Being able to ensure that a multiagent system will not generate undesirable behaviors is essential within the context of critical applications (embedded systems or real-time systems). The emergence of behaviors from the agents interaction can generate situations incompatible with the expected system execution. The standard methods to validate a multiagent system do not prevent the occurrence of undesirable behaviors during its execution in real condition. We propose a complementary approach of dynamic self-monitoring and self-regulation allowing the agents to control their own behavior. This paper goes on to present the automatic generation of self-controlled agents. We use the observer approach to verify that the agents behavior respects a set of laws throughout the system execution.

1 Introduction

The behavior of a multiagent system (MAS) emerges from the agents' interaction. Some emergent behaviors can lead the system to fail: these are **undesirable behaviors**. In our study, the autonomy is an essential specificity of cognitive agents. We consider the autonomy as the capacity of an agent to take its decision alone, without the help of another entity [1]. From the developer point of view, the agent autonomy requires to take into account the unpredictability of the agents' behavior. This view increases the possibilities of occurrence of undesirable behaviors.

The potential occurrence of undesirable behaviors could be critical for the MAS execution, especially in the context of embedded or real-time applications. To ensure the reliability of the multiagent systems, our work aims to provide a mechanism allowing the detection and the prevention of the occurrence of these undesirable behaviors, harmful to the system execution. A first approach could be the application to multiagent systems of widely used validation methods like: tests, model-checking and automatic demonstration. But, these methods cannot find all the errors in a system: the model-checking works on an abstraction of the system and its environment; the automatic demonstration is complex and heavy; tests cannot be exhaustive [3]. So, in order to detect the remaining errors, we propose to verify the system at runtime. From this perspective, the recent work of R. Paes & al. [12] proposes the use of laws to control the emergence of wrong behaviors in the context of open MAS. Their approach consists in monitoring the interaction of the agents thanks to an external entity and in describing the interaction protocols to detect violation. Similarly, the work of Klein and Dellarocas [7] proposes an approach of the exception handling thanks to a description of the exceptions and external entities filtering the agents' communication. In the work of S. Hagg [6], sentinels are used to monitor the communication between the agents, build models of other agents and intervene according to given guidelines.To our knowledge, works about the surveillance of agents behaviors use external entities to monitor the agents and detect inconsistences, particularly with respect to their communications and interactions. Moreover, most of the verification steps are done manually. The main advantage of our approach consists in the automatic creation of agents which are able to verify their behaviors from laws. This verification is based on the dynamic self-monitoring and self-regulation of each agent in order to prevent the system failure. We call this verification, the **agent control**.

The section 2 presents the agent control. The section 3 describes the laws and their use to control the agents by themselves. The section 4 details the whole process of the generation of a self-controlled agent. The section 5 describes the working of our approach and the section 6 introduces the multiagent case. Finally, the section 7 introduces a prototype implementing our approach while the section 8 concludes our paper.

2 Control of autonomous agents

The agent control is divided in three stages :

- The monitoring of the behavior of an agent or a group of agents.
- The detection of undesirable behaviors that could emerge from the execution of an agent or a group of agents.
- The regulation of the problematical agent(s).

2.1 Monitoring of an agent behavior

Monitoring the behavior of each agent in MAS in order to observe relevant events, allows to detect or prevent the occurrence of undesirable behaviors in the system [4]. Our work focuses on the observation of the MAS behavior at runtime. To observe the behavior of an agent, it is necessary to insert event detectors in the agent program. Although it is possible for a developer to insert manually the probes into the agent program, this kind of instrumentation is hard, time-consuming and prone to error [9]. We propose an automatic instrumentation of the programs of the agents, *i.e.* to insert a code of control to detect some events from the specification of *hooks* provided by the developers. The main interest of this automation is to simplify the work of the developer and to reduce the risk of errors during the instrumentation.

[1] LIP6, France, email: caroline.chopinaud@lip6.fr
[2] LIP6, France, email: amal.elfallah@lip6.fr
[3] Thales Aerospace, France, email: patrick.taillibert@fr.thalesgroup.com

2.2 Detection of undesirable behaviors

Making an exhaustive model of all the MAS behaviors is difficult and costly because of its complexity (indeterminism, state explosion, distributed nature, interleaving between the autonomous agents behaviors). So we propose to abstract this model thanks to the definition of properties about the agents' behavior. These properties are expressed using norms [13], usually defined as constraints on the agents' behavior in order to guarantee a collective order. An autonomous agent decides to respect or not a norm depending on the actions it chooses to perform. We use **laws** to describe desired or dreaded behaviors or situations ; these laws represent significant or critical requirements on the system execution and could be specified by the customer. At the opposite of norms, laws are defined independently of the agents' design and are not taken into account by the agents during the decision process. In other words, the agents can act autonomously while our approach consists in verifying afterwards if the chosen action respects the laws. Consequently the agents' implementation and the laws/control description are distinguished. The detection of undesirable behaviors can be viewed as the detection of laws transgression.

2.3 Self-regulation of agents behaviors

When an occurrence of (a potential) undesirable behavior is (to be) detected, (*i.e.* when a law is transgressed), the agents concerned by the law must regulate their behavior by themselves thanks to their capacity of reasoning. The developer provides the agents with a set of strategies of regulation associated with the transgression of laws and, at runtime, the agents deduce the behavior to follow when they are informed of a law transgression. This behavior of regulation takes the transgression into account but also the main behavior of the agent, its current state, or even the other agents. Then, it seems quite complex to define these strategies independently of the agents' implementation, as for the laws. It is the reason why, to date, the strategies are defined directly in the agent program.

3 The control description

In our approach, the description of the control applied to a multiagent system comes down to the description of laws associated with the agents.

3.1 Level of description

For a sake of generality, we would like to provide a mechanism of control which is not specific to one agent model. Hence, the laws concern abstract concepts representing the application and the model(s) of agent(s) used. We propose an ontology composed of the following concepts: AGENT, FEATURES and ACTION. The FEATURES concept has the sub-concept of Object, Message, Goal, Plan and Knowledge. The ACTION concept has the sub-concepts of Creation_Agent, Sending_Message, Receipt_Message and Migration.

The designers of the system may freely extend this set of basic concepts to refine and to enrich the description of the model(s) of the agent(s) and the application. Moreover, in order to monitor the agent behavior, we have introduced previously that the developers can describe hooks (cf 2.1). These hooks define the relation between the concepts describing the agent model(s) and its (their) implementation (for more details see [2]).

3.2 A language of Law

To express the laws, we propose a language based on deontic operators, which are widely used in the context of norms. Our language applies to events and states about the agents, corresponding to the general basic concepts of FEATURES and ACTION, introduced in the previous part. An **event** can be the execution of an action or the change of a features value. A **state** represents the resulting state of an event. The expression of time or temporal relation between the events and states is crucial in our framework and it is taken into account in our language as we will see below. Our language distinguishes two kinds of law: prohibition and obligation. A prohibition law represents the unwanted states or events of an agent. It allows the detection of situations where an event or a state which must never occur, is going to happen. An obligation law represents the expected states or events of an agent. It allows the detection of situations where an expected event or state does not occur. In our approach a law is composed of three parts:

- **CONCERNED AGENT (CA):** the statement of agents concerned by the law, i.e. the agents subject to the law and agents used to describe the law application context.
- **DEONTIC ASSERTION (DA):** the description of what is obligatory or forbidden. It is a set of relationships between an agent and an event or a state.
- **APPLICATION CONDITIONS (APC):** the description of the law context. It is an expression describing when the DA must be respected relatively to a set of events or states.

The description of our language is as follows:

LAW	:= (*CA*) (*DA* ⟨*APC*⟩).
CA	:= *agent* : AGENT ⟨ and *PROP* ⟩
DA	:= *DEON_OP EXP*
APC	:= *TEMP_OP EXP* | *TEMP_OP EXP APC*
DEON_OP	:= FORBIDDEN | OBLIGED
TEMP_OP	:= AFTER | BEFORE | IF
EXP	:= *TERM* | *TERM* AND *EXP* | *TERM* THEN *EXP*
TERM	:= *EVENT* | NOT *EVENT* | *EVENT TIME*
EVENT	:= *agent* do *SMTH* ⟨ and *PROP* ⟩ | *agent* be *SMTH* ⟨ and *PROP* ⟩
TIME	:= *-second* | *+second*

where SMTH represents a concept like ACTION or FEATURES; AGENT represents the concept of agent; PROP is a common attribute representing the properties on these concepts.

The laws are, actually, deontic formula expressed in dynamic deontic logic[10]. Knowingly, the language limits the possibilities of laws expressiveness to a sub-set of the dynamic deontic logic, allowing the description of behaviors which can be monitored. For example, a law can be expressed in this way: *"(L1): It is forbidden for an agent A1 to do an action ACT2 after an action ACT1 and before a time Sec"* and by using the language :

(L1): (A1 : Agent)
FORBIDDEN (A1 **do** ACT2) **AFTER** (A1 **do** ACT1) - Sec.

4 Self-controlled Agents

A main interest of our approach is the generation of agents being able to control their own behaviors. A self-controlled agent is obtained automatically from:

- the agent behavior program,
- the set of laws associated with the agent,
- the hooks between the concepts used in the laws and the agent model implementation.

A generated agent has a specific architecture of control allowing its monitoring and the detection of laws transgression thanks to the **observer mechanism** [5].

4.1 The observer

The observer mechanism is used in the context of real-time systems testing, on-line validation of parallel, or even distributed systems. The observer consists in a parallel execution of a program and a model of properties applied to the program execution. The model and the program are connected with control points. A controller checks on the model and the program execution are consistent. If the system execution does not match the model, the verification fails.

We propose to put this approach in place into the agents in order to provide them with the means to control their own behavior. A law is modeled by means of a Petri net [11] whose transitions are bound to the program with control points. The Petri nets representing each law are automatically generated (c.f. 4.3). We insert into the agent behavior program the control points and we generate a runnable agent with a specific architecture, using the observer approach. When the program execution meets a control point, the controller makes sure the tokens are in the right place at the right time in the corresponding net, and brings about some change in the model, accordingly.

4.2 The agent architecture

A target generated agent has a specific architecture divided into two parts, the *behavior part* and the *control part*. The behavior part includes the real agent behavior and strategies of regulation defined by the developer. Indeed, we would like not only that the verification fails when an inconsistency is detected, but that the agent regulates its behavior whenever a law is transgressed. The control part includes the set of Petri nets representing the laws associated with the agent and makes sure of the detection of the laws violation. The connections between the program and the models are effective thanks to a sending of information from the behavior part to the control part. To allow this sending of information, we instrument the behavior part by inserting automatically some control points associated with the events and states contained in the laws. The control part receives the information and verifies the respect of the laws.

4.3 The generation

The generation of a self-controlled agent comes down to the generation of the Petri nets representing the laws concerning the agent and the instrumentation of the agent behavior to detect the occurrence of events and states expressed in the laws.

4.3.1 The instrumentation

We propose an automatic instrumentation of the agent behavior program to monitor the occurrence of the events and states, defined in the laws, by means of **control points**. This instrumentation is done thanks to the hooks defined by the developer, between the concepts describing the model and its implementation. In order to do that, we draw our inspiration from the principle of weaving. The **weaving** is an important part of aspect programming [14]. The latter uses the weaving to inject aspects in classes of an application, at the methods level, to modify the system execution after the compilation. An aspect is a module representing crosscutting concerns. The interest of aspect programming to integrate the monitoring in an application was demonstrated in another light by [8]. So, our approach could be summarize as follows:

1. Extracting the events or states to be detected.
2. For each event or state, searching for the hook that has been provided by the developer.
3. Injecting, before or after that hook, the piece of code allowing to send the information to the control part and to recover of possible information of transgression. The lattest enables the agent to start a strategy of regulation.

4.3.2 The generation of Petri nets

The generation of a Petri net representing a law is divided into three stages:

1. The translation of the law in a logic expression L, in order to point out a set of atomic expressions, $\{e_1, ..., e_n\}$.
2. Incrementally:
   (a) The deduction of a set of Petri nets, $\{p_1, ..., p_n\}$ representative of each expression in $\{e_1, ..., e_n\}$.
   (b) The merging of all the nets in $\{p_1, ..., p_n\}$ from the relations between $e_1, ..., e_n$ to obtain a final Petri net, P, representing the law.

To perform the step *(1)*, each operator in the language has a meaning in dynamic deontic logic. We use a set of translation rules to obtain the logic expression representing the law (cf. Set of Rules 1). For the step *(2)*, we propose to represent atomic expression by means of a Petri net, as in the set of rules 2; and merging rules, as described in the set of rules 3. To express a prohibition we use an *inhibitor arc* and to express a time, we use a *temporal Petri net*. The final Petri net P representing the law, is embedded into the control part of each agent submitted to the law.

Set of Rules 1

Let Act be a set of actions.
Let $Assert$ be a set of assertions.
Let $State$ be a set of states included in Assert.
$\forall \alpha \in Act, \phi \in Assert, \beta \in State.$
(1) $FORBIDDEN\,\alpha \equiv F\alpha$
(2) $OBLIGED\,\alpha \equiv O\alpha$
(3) $\phi\, AFTER\, \alpha \equiv [\alpha]\phi$
(4) $FORBIDDEN \alpha_1\, BEFORE\, \alpha_2 \equiv done(\alpha_2) \vee F\alpha_1$
(5) $OBLIGED \alpha_1\, BEFORE\, \alpha_2 \equiv \neg done(\alpha) \vee O\alpha_1$
(6) $\alpha_1\, AND\, \alpha_2 \equiv (\alpha_1; \alpha_2) \cup (\alpha_2; \alpha_1)$
(7) $\alpha_1\, THEN\, \alpha_2 \equiv \alpha_1; \alpha_2$
(8) $\phi\, IF\, \beta \equiv \beta \supset \phi$
(9) $FORBIDDEN \alpha\ -\ Sec \equiv done(time(Sec)) \vee F\alpha$
(10) $OBLIGED \alpha\ -\ Sec \equiv \neg done(time(Sec)) \vee O\alpha$
(11) $\phi\ +\ Sec \equiv [time(Sec)]\phi$

Set of Rules 2

Let $PN =< P, T, Pre, Post >$ be a Petri net, where:
P: a set of places, $P = (p_1, p_2, ..., p_n)$,
T: a set of transitions, $T = (t_1, t_2, ..., t_n)$,
Pre: a function, $P \times T \rightarrow N$, which defines directed arcs from places to transitions. (Note Pre, when the directed arc is an inhibitor arc)*
Post: a function, $P \times T \rightarrow N$, which defines directed arcs from transitions to places
Let t_α be the transition associated with the action α.
In the following, the rule $X \Rightarrow Y$ means that X is a logical assertion and Y the corresponding Petri net.
$\forall \alpha \in Act, \beta \in State$
(1) $F\alpha \Rightarrow < (p_i, p_j), t_\alpha, Pre^*(p_i, t_\alpha), Post(p_j, t_\alpha) >$
(2) $O\alpha \Rightarrow < (p_i, p_j), t_\alpha, Pre(p_i, t_\alpha), Post(p_j, t_\alpha) >$
(3) $done\alpha \Rightarrow < (p_i, p_j), t_\alpha, Pre(p_i, t_\alpha), Post(p_j, t_\alpha) >$

(4) $\neg done\alpha \Rightarrow < p_i, p_j, t_\alpha, Pre^*(p_i, t_\alpha), Post(p_j, t_\alpha) >$
(5) $\beta \Rightarrow < (p_i, \beta, p_j), t_i, t_j, (Pre(p_i, t_i), Pre(\beta, t_j)), (Post(\beta, t_i), Post(p_j, t)) >$
(6) $\alpha \Rightarrow < (p_i, p_j), t_\alpha, Pre(p_i, t_\alpha), Post(p_j, t_\alpha) >$

Set of Rules 3
Let t_α be the transition associated with the event α.
Let t_ϕ be the transition associated with the event used in ϕ.
Let Pre(p,t) be the function which returns the input place p of a transition t.
Let Post(p,t) be the function which returns the output place p of a transition t.
Let $merge_p(p_1, p_2)$ be the operator of fusion of two places p_1 and p_2.
$\forall \alpha \in Act, \phi \in Assert, \beta \in State$
(1) $[\alpha]\phi \Rightarrow merge_p(Post(p, t_\alpha), Pre(p, t_\phi))$.
(2) $\phi_1 \vee \phi_2 \Rightarrow merge_p(Pre(p, t_{\phi_2}), Pre(p, t_{\phi_1}))$
(3) $\phi_1 \wedge \phi_2 \Rightarrow separation\, in\, two\, Petri\, nets: PN_{\phi_1}\, and\, PN_{\phi_2}$
(4) $\beta \supset \phi \Rightarrow merge_p(Pre(p, t_\phi), \beta)$

5 The detection of transgression

We remind that the behavior part sends information about its states and events to the control part. From this information, the control part must verify if the laws associated with the agent are respected. To detect the laws violation, the control part uses the two following algorithms:

Algorithm 1 In the main part of the control part

1: Let I be an information about the agent behavior.
2: Let $\{t_1, ..., t_n\}$ be the set of transitions associated with I.
3: Let $\{P_1, ..., P_m\}$ be the set of Petri nets associated with the agent.
4: Let $\{Pact_1, ..., Pact_p\}$ be the set of activated Petri nets (i.e. associated with the laws to be manage)
5: Let t_{ij} be the transition i of the net j.
6: **for all** $P_k \in \{P_1, ..., P_m\}$ with $t_{1k} \in \{t_1, ..., t_n\}$ **do**
7: $Pact_{p+1} \leftarrow$ create an instance of P_k
8: add $Pact_{p+1}$ in $\{Pact_1, ..., Pact_p\}$
9: **end for**
10: Let $\{Pact_1, ..., Pact_l\}$ be the set of the activated Petri nets including a $t_{ij} \in \{t_1, ..., t_n\}, j \in \{1, ..., l\}$
11: **for all** $Pact_j \in \{Pact_1, ..., Pact_l\}$ **do**
12: inform $Pact_j$ of the information associated with t_{ij}
13: **end for**

Algorithm 2 Within the instances of Petri net

1: Let $\{t_1, t_n\}$ be the set of transitions of the Petri net.
2: Let I be the sent information.
3: Let t_I be the transition associated with the information I $\in t_1, ..., t_n$.
4: A transition t_i is *activated* if a token stands in all the previous places of t_i (in our Petri net the arcs are one-valuated).
5: **if** t_I is activated **then**
6: **if** t_I is firable **then**
7: fire the transition t_I
8: **else**
9: throw exception
10: **end if**
11: **end if**

The control part receives the information and generates a new instances of each Petri net beginning by the transition associated with the information. The monitoring of the law, *i.e.* the associated Petri net, is **activated**. Then, the control part forwards the information to all the instances where the information is expected. If the transition associated with the information is activated , the instance verifies if the transition is firable (according to the line 6, algo 2) and changes the Petri net. When an inconsistency is detected, the instance throws an exception and the control part warns the behavior part by sending information of transgression. An instance is destroyed when the net is in a final state, so when the law is respected or when the delay of the law observation is elapsed.

When the behavior part sends an information to the control part, it is stopped until the control part permits its restarting. The temporary deadlock is essential if we want to prevent the execution of forbidden actions.

6 About the multiagent behaviors

In the previous parts, we have presented the whole functioning of the self-controlled agents. Particularly, we tackle a single agent view, that is when a law affects only one agent. Even if this first part solves a set of problems related to individual agent behavior, we think that shifting to the multiagent behaviors rises other problems. Indeed, the question is the control of the multiagent system behavior when several agents, individually correct, are put together. A first step to answer this question is the handling of the control when a law is applied to several agents.

Our aim is to distribute as much as possible the control into each agent involved in the law. We would like to avoid a centralized solution. We emphasize three points to put forward in that case:

- How is the distribution of the Petri net done among all the agents concerned by the law?
- How do the control parts of the agents collaborate to detect a law transgression?
- How is the regulation done? Who is the culprit?

6.1 The net distribution

The Petri net representing a law applied to several agents is deduced as in a single agent context. Then, the net is distributed into the control parts of the agents concerned by the law. This distribution is done by following the algorithm 3.

Algorithm 3 Petri net Distribution

1: Let L be a law.
2: Let P be the Petri net representing L.
3: Let $Nbagent$ be the number of agents concerned by L.
4: Let $\{T_{m1}, ..., T_{mn}\}$ be the set of transitions where the information associated with t_{mi} comes from the agent $AG_m, m \in \{1, ..., Nbagent\}$.
5: Let $Ppre_{mt}$ be the input place of the transition t of the agent AG_m.
6: Let $Ppost_{mt}$ be the output place of the transition t of the agent AG_m.
7: **for all** $T \in \{T_{m1}, ..., T_{mn}\}$ **do**
8: put T in the control part of the agent AG_m.
9: **end for**
10: **if** $Ppost_{mt} \equiv Ppre_{(m+1)t'}$ **then**
11: put $Ppost_{mt}$ in the control part of the agent $AG_{(m+1)}$.
12: **end if**
13: **if** $Ppre_{(m+1)t'} \equiv Ppre_{mt}$ **then**
14: put $Ppre_{mt}$ in the control part of the agent AG_m.
15: **end if**
16: **for all** $t' \in \{1, ..., n\}$,$m' \in \{1, ..., Nbagent\}$ **do**
17: **if** $Ppost_{mt} \not\equiv Ppre_{m't'}$ **then**
18: put $Ppost_{mt}$ in the control part of the agent AG_m.
19: **end if**
20: **if** $Ppre_{mt} \not\equiv Ppost_{m't'}$ **then**
21: put $Ppre_{mt}$ in the control part of the agent AG_m.
22: **end if**
23: **end for**

Let us note that the control parts are only linked through the arcs between places and transitions (themselves distributed over the control parts of agents). These links represent the information flow between the control parts (*i.e* the flow of the token).

6.2 The collaboration

A control part is willing to listen to the behavior part of its agent and to the control part of the other agents. Indeed, when a law is distributed into several agents, the control parts of these agents are able to exchange information about the occurrence of events and states. We have seen that the net distribution between the control parts is in such a way that only branches $Transition \rightarrow Place$ or $Place \rightarrow Transition$ link every pieces of the net. These arcs represent the flow of an information between the agents. In order to explain the collaboration between the control parts, we propose to use an example.

So, when a control part, C_A, receives an information from its agent, if this information is associated with a transition T whose the next place is in the control part of another agent, C_B, then C_A sends information about the firing of this transition, (actually, it sends the token) to the control part C_B and waits for an acknowledgment of receipt. During this waiting, the execution of the agent is temporarily stopped and the information associated with T is considered as always available. The control part C_B receives the information, sends the acknowledgment to the control part C_A and executes the algorithm 1, as if the information comes to its associated behavior part. When C_A receives the acknowledgment, the transition can be really fired, the information associated with T is consumed and the agent behavior can continue.

6.3 The regulation

The detection of a law transgression generates an exception in the control part. This exception generates the transmission of information of transgression from the control part to the behavior part of the agent. In the context of a law applied to several agents, the mechanism remains the same. All the control parts detecting a transgression, send the exception to the associated behavior part. So, we suppose that the strategies of regulation solve the problem of who is the culprit. Putting such strategy in place in the agents is not trivial and will be treated in our future work.

7 Implementation

Our approach is implemented in a framework: SCAAR. This framework provides the means to describe laws, concepts... and to generate self-controlled agents. A prototype of SCAAR has been implemented in SICStus Prolog. This first version provides a part of the language, to describe single agent laws and the generation, from the laws description, of the Petri nets. The net distribution into several agents is in progress. The behavior of the control part has been implemented using the algorithms described in the section 5. The generated self-controlled agents are made up of two Prolog processes, one executing the real agent behavior, one executing the control part behavior. These two parts communicate by TCP/IP channel for the messages passing about the occurrence of events in the agent behavior and the detection of laws transgression.

Our approach has been applied to a multiagent application of mobile naval targets localization, developed in our service: *Interloc*. In Interloc, planes seek to detect boats, in a passive way. The system is implemented in Prolog. In order to demonstrate the robustness of MAS application, Interloc provides a set of wrong agent behaviors, like:

- Dumb: The agent sends no message.
- Deaf: The agent reads no message.
- Selfish: The agent always refuses the requested services.
- Absurd: The agent accepts the services but sends absurd results.
- Latecomer: The agent is always late replying.

Therefore, Interloc is an interesting testbed to validate our approach. We have expressed laws to detect wrong behaviors and the application has been executed with modified agents in order to detect these "undesirable behaviors", by using the generator provided by SCAAR.

8 Conclusion

We have presented the principles of the autonomous agent control. Particularly, we have described the mechanisms and the architecture of the self-controlled agents, agents able to monitor their own behaviors. This monitoring is done from the specification of the laws that the agents must respect during their execution. A distributed solution to control multiagent behaviors has been proposed. An advantage of our approach is the possibility to define laws at a high level, allowing its application to several models of agents. Another point is the automatic generation of the agents and the simplification of the developer's work to take the control into account. The framework SCAAR, allowing the generation of self-controlled agents has been introduced. Our future work is essentially to enhance the implementation and treat more carefully the regulation part.

REFERENCES

[1] K.S. Barber and C.E. Martin, 'Agent autonomy : Specification, measurement and dynamic adjustment', in *Proc. of the Autonomy Control Software workshop at Autonomous Agents'99*, pp. 8–15, (May 1999).

[2] C. Chopinaud, A. El Fallah Seghrouchni, and P. Taillibert, 'Dynamic self-control of autonomous agents', in *Post-Proceedings of PROMAS'05, LNAI 3862, Spinger Verlag*, (2006).

[3] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, MIT Press, 2000.

[4] M. de Sousa Dias and D.J. Richardson, 'Issues on software monitoring', Technical report, Department of Information and Computer Science, University of California, (July 2002).

[5] M. Diaz, G. Juanole, and J-P. Courtiat, 'Observer-a concept for formal on-line validation of distributed systems', *IEEE Trans. Softw. Eng.*, **20**(12), 900–913, (1994).

[6] Staffan Hagg, 'A sentinel approach to fault handling in multi-agent systems', in *Revised Papers from the Second Australian Workshop on Distributed Artificial Intelligence: Multi-Agent Systems: Methodologies and Applications*, volume 1286 of *LNCS*, pp. 181–195. Spinger-Verlag, (1996).

[7] Mark Klein and Chrysanthos Dellarocas, 'Exception handling in agent systems', in *In proceedings of Agents'99*, pp. 62–68, (1999).

[8] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat, 'Program instrumentation for debugging and monitoring with AspectC++', in *Proc. of the 5th IEEE International symposium on Object-Oriented Real-time Distributed Computing*, Washington DC, USA, (April 29 – May 1 2002).

[9] M. Mansouri-Samani, *Monitoring of Distributed Sytems*, Ph.D. dissertation, University of London, London, UK, 1995.

[10] JJCH. Meyer, 'A different approach to deontic logic: deontic logic viewed as a variant of dynamic logic', *Notre dame journal of formal logic*, **29**(1), 109–136, (Winter 1988).

[11] T. Murata, 'Petri nets: Properties, analysis and applications', *In Proc. of IEEE*, **77**(4), 541–580, (April 1989).

[12] Rodrigo Paes, gustavo Carvalho, carlos Lucena, Paulo Alencar, Hyggo Almeida, and Viviane Silva, 'Specifying laws in open multi-agent systems', in *ANIREM*, Utrecht, (July 2005).

[13] J. Vázquez-Salceda, H. Aldewerld, and F. Dignum, 'Implementing norms in multiagent systems', in *Proceedings of MATES'04*, Erfurt, Germany, (September, 29–30 2004).

[14] Dean Wampler. The future of aspect oriented programming, 2003. White Paper, available on *http://www.aspectprogramming.com*.