Dynamic Self-control of Autonomous Agents

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

[1] Thales Airborne Systems, 2 avenue Gay Lussac 78851 Elancourt, France
{caroline.chopinaud, patrick.taillibert}@fr.thalesgroup.com
[2] LIP6, 8 rue du Capitaine Scott 75015 Paris, France
amal.elfallah@lip6.fr

Abstract. Being able to trust in a system behavior is of prime importance, particularly within the context of critical applications as embedded or real-time systems. We want to ensure that a multiagent system has a behavior corresponding to what its developers expect. The use of standard techniques to validate a system does not guarantee it against the occurence of errors in real condition of execution. So, we propose an additional approach of dynamic self-monitoring and self-regulation such that an agent might control, in real condition, its own behavior. Our approach consists in providing the agents with a set of laws that they have to respect throughout their execution. This paper presents a framework which generates agents capable of self-control from an agent model, a behavior description and laws. For that, the framework modifies the agents program by injecting, some checkpoints allowing the detection of particular events. The laws are represented in the agents by Petri nets connected to the checkpoints in order to verify the agreement between their behavior and the laws. The principles of the framework are illustrated on an example.

1 Introduction

Autonomy is an essential feature of cognitive agents. We will consider the autonomy as the ability of an agent to take its decisions without the help of another entity [1]. From the developer's point of view, it means that the implementation of an agent requires to take into account that the behavior of the other agents cannot be predicted with certainty. This perspective brings up the problem of the confidence that we can have in a system behavior. When critical applications are concerned, the use of such system might raise objections because of this unpredictability. So, it is essential to ensure that MAS and its agents respect some behavioral requirements which are essential for the application.

The aim of our research is to ensure that a MAS behavior will fulfill with these requirements. A first approach could be the use of classical methods of validation, such as tests, Model Checking [2] and automatic demonstration to validate a multiagent system. But, these techniques are never in the position to detect all possible errors and let situations in which errors may occur at runtime. That is the reason why we will consider an on-line verification of the system behavior.

R.H. Bordini et al. (Eds.): ProMAS 2005, LNAI 3862, pp. 41–56, 2006.
© Springer-Verlag Berlin Heidelberg 2006

This verification consists in monitoring and regulating the system behavior to prevent its failure. We call this verification the **agent control**.

Moreover, we think that the agents are better placed to make themselves the control of their behavior. So, we provide the agents with the means to monitor their behavior and, thanks to their capabilities of reasoning, they can regulate their behavior in order to avoid **undesirable behaviors**.

Although it is possible for a developer to insert the control code into the agents, a manual instrumentation of a system program, to insert probes, is hard, time-consuming and prone to error [12]. When several agents are concerned, it is worst. So, upgrading the agents behavior and control becomes hard, if we consider that the monitoring code is fragmented in the agent program and also distributed among the agents. On the basis of automatic instrumentation for monitoring distributed systems, a possible solution could be to automatically modify the agent program in order to introduce the control and so, facilitate the work of the developer. We are particularly interested in monitoring software which consists in inserting software probes into the program to detect events [3]. The automation of the insertion can take two forms : (1) the developer uses a metalanguage [10] or a routines library [6] allowing the insertion of probes in a transparent way; (2) the insertion will be made by compiler from the specification of the interested events [9]. We focus on the last form and we propose a generator which creates agents being able to check their own behavior from a description of the requirements associated with the agents and their behavior program.

In the section 2 we will present the principles of the control of autonomous agent. An example will be described in the section 3 to illustrate during the rest of the paper the operation of the generator. In the section 4, we will describe the SCAAR framework allowing the generation of self-controlled agents. In this part we will focus on laws concepts and generation of the control from these laws.

2 Control of Autonomous Agent

We consider controlling MAS relies on the monitoring of agents behavior during the MAS execution and also the regulation when an error occurs. The idea it to make an automatic insertion into the agents of the necessary means of control, allowing agents to monitor themselves and detect undesirable behaviors.

2.1 Behavior Verification

Making a model of a whole MAS and its agents is not conceivable because of its complexity (indeterminism, state explosion, distributed nature). So, we propose to use norms to express properties about the agents behavior. In general, norms [15] define constraints on the agents behavior in order to guarantee a social order. An agent decide to respect or not a norm by restricting its set of possible actions. We use **laws** to describe desired or dreaded behaviors or

situations. Laws are norms that don't be taken into account by the agents at the decision process (*i.e.* the agents can act as they want and our approach consists in verifying if the chosen action respects the laws afterwards) because we want to distinguish the agent implementation and the laws/control description. The laws represents signification or critical requirements of the system execution. An agent capable of self-control checks that laws are respected throughout its execution. But monitoring is not enough, when an agent detects the transgression of a law, it must regulate its behavior from transgression information.

In oder that the agent can deduce their behavior when they are informed of a law transgression, we suppose that the laws are known by the agents. Either the developer constructs the agents from requirements, consequently he verifies that the agents respect the laws by construction, or the agents have a representation of this laws that they attempt to respect throughout their execution. So our approach consists in adding a dynamic verification to make sure that the developer correctly implements the agents and the latter always respect the laws once running in multiagent context.

2.2 Level of Laws

We wish that the person who decides of the laws does not be necessarily the developer. In general the customers define the requirements and the developers implement the system form these requirements. Also, the customer is in position to know what is important to verify without to know the agents implementation. So, we suppose that the laws are expressed in natural language by the customer and translated by an expert in a description language. Therefore, it is necessary to express the laws at an abstraction level understandable by the customer and allowing an easy translation.

Moreover, for a sake of generality, we would provide a control mechanism for several agent models. The level of law must permit to include several kinds of agent models. So, the laws must state general **concepts** representing the agent model and the application. The model designer provides a set of concepts representing the model specificities and the system designer/developer provides typical application concepts. From this set of concepts, the expert can describe the laws expressed by the customer by using the description language.

2.3 Control Enforcement

The control enforcement is divided in five steps (Fig.1):

1. The model developer must provide a description of the concepts and their hooks with the model implementation.
2. The customer provides the set of laws that he wants to be verified throughout the system execution. These laws can concern one or several agents.
3. The system designer/developer describes the concepts representing the application.The system developer implements the agents form the set of laws or/and the agents use the laws to deduce their behavior at runtime. He provides the agents with strategies of regulation associated with each laws.

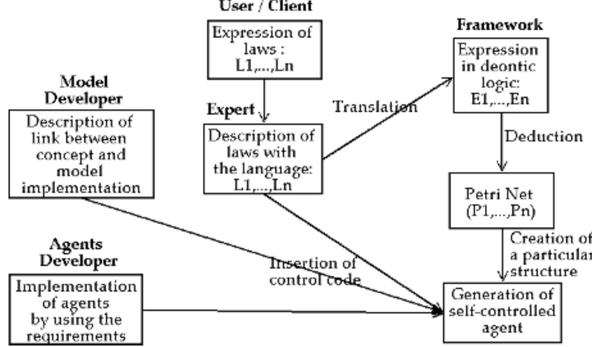

Fig. 1. Aspects of agent control

4. An expert translates the laws expressed in (2) by using a description language. The laws state concepts describes in (1) and (3).
5. The generator translates laws, written in a particular language, into expressions using deontic logic. From these expressions, the generator deduces the Petri nets representing the laws. Also, the generator inserts control points into the agents program to allow the detection of the expected events defined in the laws. The generator provides the agents with a particular structure to obtain self-controlled agents.

3 An Example

3.1 The Multiagent System

We introduce a simple multiagent system of problem resolution. MAS is constituted of three kinds of agent : an agent A, used as an interface between the user and the system; an agent B, assuming the management of the problems forwarded by A; an agent C, resolving the problems sent by B. The agent model used is a Petri net. The behavior of each agent is described in figure 2.

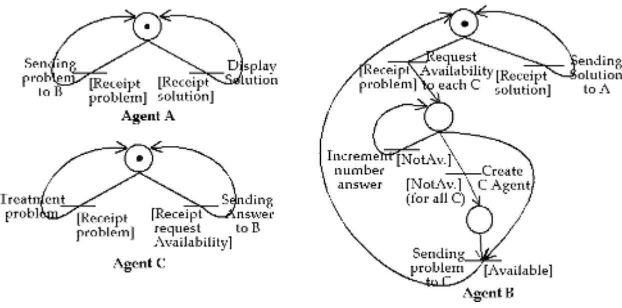

Fig. 2. The behavior of each agents

3.2 Laws of the System

The customer expresses the laws of the system in English. We will take the example of three laws which can be associated with the previous MAS :

L1: "an A agent must not send message to a B agent at a rate greater than one message per second". For instance, this law ensures that the agent B can follow the requests sent by the agent A and C can answer in a reasonable time.
L2: "a B agent must not create a C agent if another C agent is available in the system". This law prevents the agent B to create too many agents and consequently prevents the application overload.
L3: "a C agent must receive consecutively a message about the waiting time, a message of identification, the data of the problem without any other interleaved messages". This law allows the verification of communication protocols used between the agents. In this case, the law concerns the receiver agent which must receive three messages in a particular order without another message between them.

4 SCAAR: A Framework for the Generation of Self-controlled Autonomous Agents

SCAAR (Self-Controlled Autonomous Agent geneRator) is a framework allowing the generation of agents capable of self-control. The generator uses the set of laws associated to the agents, the set of concepts used in laws and injects the control code into the agents program. The figure 3 represents the framework architecture which consists of :

- **Ontology:** The set of concepts representing the agents models and the application.
- **Laws:** The set of properties that the agents must respect.
- **Agent Model:** The hooks between the model description (the concepts) and the model implementation.
- **Agent Program:** The code of agent behavior.
- **Generator:** The creator of the final agent from the previous elements.
- **Self-controlled agent:** The executable final agent. It monitors its own behavior in order to verify if the laws are respected and regulate it if a law is transgressed.

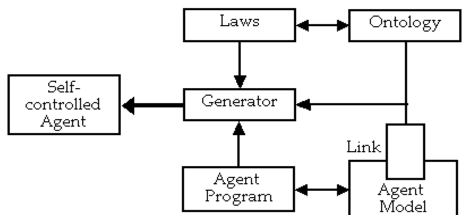

Fig. 3. The SCAAR Framework

4.1 Ontology

We saw that the laws are based on high-level concepts allowing the description of agents and system specificity. So, we construct a basic ontology representing known models used to construct agents. A set of agent concepts is provided by D.N. Lam and K.S. Barber [8] for agent verification. The concepts proposed are: *Goal*, *Belief*, *Intention*, *Action*, *Event*, *Message*. We take a part of this concepts (*Goal*, *Action*, *Message*) and we add other ones which are more typical of agent models from our point of view (BDI, CLAIM, personal agent models): *Agent*, *Object*, *Knowledge*, *Plan*, *Agent Creation*, *Message Sending*, *Message Receipt*, *Migration*). We propose to divide the concepts up into three categories: AGENT, textscFeature and ACTION. The figure 4 represents the distribution of the concepts.

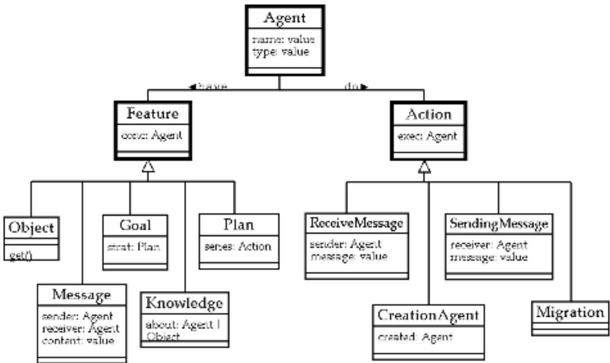

Fig. 4. Agent Ontology

This set of basic concepts can be extended by the model designer to refine the model description with sub-concepts or instances. The system designer can extend the set of concepts defining the model with instances. In our example, the agent model can be described with the basic concept of: `ReceiptMessage`, `SendingMessage`, `CreationAgent`, `Action`, `Agent`. The refinement of the description is done with the concepts of: `State` which is a sub-concept of `Knowledge` and has a parameter : available or not, and `ReceivedMessage` which is a sub-concept of `Message`.

4.2 The Laws

We can distinguish two kinds of law :

- One representing unwanted state or behavior of an agent. It allows the detection of situations where an event occurs while it should not.
- One representing expected state or behavior of an agent. It allows the detection of situations where an expected event does not occurs.

We propose that our laws are expressed by using deontic operators, which are widely used in the context of norms. So, we provide a language of law description allowing the expression of prohibition and obligation to represent the two previous kinds of law. The language applies to events and states about the agents, corresponding to the general basic concepts of FEATURES and ACTION introduced previously. An **event** can be the execution of an action or the change of a feature value. A **state** represents the resulting state of an event. The expression of time or temporal relation between the events and states is possible. We can divide a law in three parts:

- **CONCERNED_AGENTS (CA)**: The statement of the agents concerned by the law. These are the agents that can be subject to the law and agents used to describe the laws application context.
- **DEONTIC_ASSERTION (DA)**: The description of what is obligatory and forbidden. It is a set of relationship 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.

So, the language syntax is as follows:

LAW	:= (CA)(DA ⟨APC⟩)
CA	:= (*agent* : AGENT ⟨ **and** PROP⟩)+
DA	:= DEONTIC EXP | DA **AND** EXP | DA **THEN** EXP
APC	:= QUA1 EVENT ⟨ **AND** EVENT⟩ | QUA1 EVENT ⟨ **THEN** EVENT⟩ | QUA2 STATE ⟨ **AND** STATE⟩ | QUA1 *seconde* | APC APC
DEONTIC	:= **FORBIDDEN** | **OBLIGED**
EXP	:= (EVENT) | NOT EXP
EVENT	:= *agent* **do** SMTH ⟨ **and** PROP⟩
STATE	:= *agent* **be** SMTH ⟨ **and** PROP⟩
QUA1	:= **AFTER** | **BEFORE**
QUA2	:= **IF**
PROP	:= *funct(concept.argument)*
AGENT	:= *Concept : Agent*
SMTH	:= *Concept : Action* | *Concept : Feature*

The semantic of our language is based on the dynamic deontic logic [13]. This logic is a variant of the deontic logic [16] and allows the expression of relation of time between actions and states. The table 1 represents the correspondence between the language of laws description and the dynamic deontic logic.

To explain the language we will express the law given in section 3[1].

[1] An agent of type X will be denoted agX.

Table 1. Relation between our language and the Dynamic Deontic logic (not exhaustive)

LANGUAGE	**LOGIC**
FORBIDDEN (EV)	F(EV)
OBLIGED (EV)	O (EV)
FORBIDDEN (EV1) AND (EV2)	F(EV1 & EV2)
FORBIDDEN (EV1) THEN (EV2)	F(EV1 ; EV2)
FORBIDDEN (EV2) AFTER (EV1)	[EV1]F(EV2)
FORBIDDEN (EV1) BEFORE (EV2)	F(EV1)[EV2]
... AFTER (EV1) AND (EV2)	[EV1 & EV2]...
... AFTER (EV1) THEN (EV2)	[EV1 ; EV2]...
FORBIDDEN (EV1) IF (STATE)	STATE $\supset$ F(EV1)
NOT(EVENT)	$\overline{EVENT}$
EV1 EV2 BEFORE(Sec)	EV1 EV2[time(Sec)]
EV1 EV2 AFTER(Sec)	EV1[time(Sec)]EV2

L1 : "an A agent must not send message to a B agent at a rate greater than one message per second" could be expressed as a prohibition:

(CA) For each agA.
(DA) It is forbidden for agA to send a message to agB,
(APC) After agA send a message to agB and before one second.

By using the language and the concepts defined for the system, we can write the law as:

(L1) (agA : Agent and agA.type = A)(agB : Agent and agB.type= B)
FORBIDDEN(agA **do** SendingMessage **and** receiver = agB)
AFTER(agA **do** SendingMessage **and** receiver = agB)
BEFORE(1).

L2 : "a B agent must not create a C agent if another C agent is available in the system" could be expressed as a prohibition:

(CA) For each agB and for each agC.
(DA) It is forbidden for agB to create a new C agent,
(APC) If agC is in an available state.

By using the language:

(L2) (agB : Agent **and** agB.type = B)(agC : Agent **and** agC.type = C)
FORBIDDEN(agB **do** CreationAgent **and** created.type = C)
IF(agC **be** State **and** value = available).

L3 : "a C agent must receive consecutively a message about the waiting time (M1), a message of identification (M2), the data of the problem (M3) without any another interleaved message" could be expressed as an obligation:

(CA) For each agC.
(DA) It is obligatory for agC to receive M1 then M2 then M3 without any other interleaved message,
(CAP) nothing.

By using the language:

(L3)(agC : Agent **and** agC.type = C)
OBLIGED(agC **be** ReceivedMessage **and** content = M1)
THEN (agC **do** ReceivedMessage **and** content = M2)
AND NOT (agC **do** ReceivedMessage **and** content <> M2)
THEN (agC **do** ReceivedMessage **and** content = M3)
AND NOT(agC **do** ReceivedMessage **and** content <> M3).

4.3 Hooks Between Concepts and the Implementation of the Agent Model

The generation of self-controlled agents required the agents program instrumentation in order to insert control points for verifying the respect of the laws. The concepts used in laws must have a representation in the agents programs. The designer of the model provides the hooks between the abstract concepts and the implementation of the agent model. From this hooks the generator inserts the control code in the implementation of the agent model and consequently in the agents.

```
hook('SendingMessage', predicate(sendMessage, 2),
      [MESSAGE, RECEIVER], [argument(1), argument(2)]).

hook('ReceivedMessage', argument(predicate(setIncomingMessage,2),1),
     [CONTENT, SENDER],
     [call(predicate(getContent,1),1), call(predicate(getSender,1),1)]).
```

Fig. 5. Example of hooks between concepts and model program

Let's see on the example the necessary hooks to insert control in the agents. The agents are programmed in Prolog, we focus on two concepts: `SendingMessage` and `ReceivedMessage`. The first concept corresponds, in the implementation of agent model, to the *sendMessage* clause. The message can be found in the first argument and the receiver in the second argument of *sendMessage*. `Received-Message` concept is linked in the program with a certain variable. We get its value in argument of the clause *SetIncomingMessage*. The message content and the sender can also be get by a call of a method. The figure 5 shows the code to describe the hooks between the concepts and the program.

4.4 Structure of Generated Agents

A generated agent is obtained directly from :

- The agent behavior program
- The set of laws associated with the agent
- The links between the concepts used in the laws and the model implementation

To allow the agent to control their own behavior, we propose the use of the **observer approach** [4].

The Observer Approach. The observer approach consists in executing a program and a model of property, about the program execution, in parallel. The model and the program are connected with control points. A controller checks on the model and the program execution are consistent.

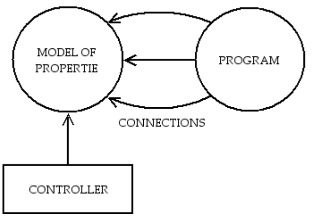

Fig. 6. The observer approach

For instance, the properties can be modeled in the form of Petri net whose transitions are bound to the program with the control points. When the program execution finds a control point, the controller makes sure the tokens are in the right place at the right time in the corresponding nets, and brings about some change in the model, accordingly. If the system execution does not match to the models, the verification fails.

So, we propose to put this approach in place into the agents in order to provide them the means of controlling their own behavior. Firstly, the laws are modeled by Petri nets. In order to simplify this stage of modeling, we propose to generate automatically the Petri nets representing the laws. Secondly, we insert into the agent behavior program the control points linked to the transitions of the Petri nets and we generate a runnable agent with a specific architecture, using the observer approach.

The Architecture. A generated agent has a specific architecture allowing the monitoring of the agent behavior and the detection of the transgression of the laws associated with an agent. The architecture is divided in two parts, the behavior part and the control part. The figure 7 represents the agent architecture.

The behavior part matches the program under surveillance in the observer approach. It includes the real agent behavior and strategies of regulation defined

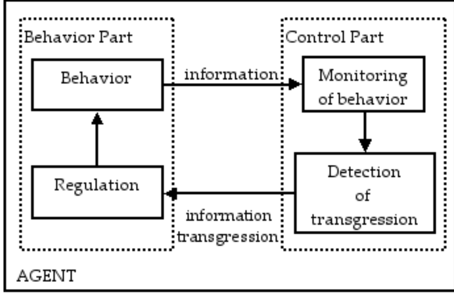

Fig. 7. The architecture of a self-controlled agent

by the developer. Indeed, we would not like only that the verification fails when an inconsistence is detected but that the agent can regulate its behavior when a law is transgressed. The control part matches the controller which is an integral part of the agent. 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 simulated by 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.

4.5 The Generator

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

The Instrumentation. To monitor a system execution, it is essential to insert probes into the program to detect the occurrence of events. We propose an automatic instrumentation of the agent behavior program to monitor the occurrence of the events and states expressed in the laws by inserting **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 ours inspiration from the principle of weaving. The **weaving** is an important part of the aspect programming [17]. The latter consists in modularizing crosscutting structure. The aspect programming uses the weaving to inject aspects in classes of an application, at methods level, to modify the system execution after the compilation. An aspect is a module representing crosscutting concerns. The interest of the aspect programming to integrate the monitoring in an application was demonstrated in another light by [11]. So, our approach consists in:

Table 2. Translation of logic expression in Petri Net (not exhaustive)

LOGIC	PETRI NET	LOGIC	PETRI NET
F(EVENT)		O(EVENT)	
F(EV1 & EV2)		O(EV1 & EV2)	
F(EV1 ; EV2)		[EV1]F(EV2)	
F(EV1)[EV2]		[EV1 & EV2]	
[EV1 ; EV2]		STATE $\supset$ F(EV1)	
EV1 EV2 [Sec]		EV1 [Sec] EV2	

1. Extracting the events to be detected.
2. For each event, searching for the provided hook to the implementation.
3. Injecting, before or after the provided hook, the code allowing the sending of information to the control part and the recovery of possible information of transgression to enable the agent to begin a strategy of regulation.

The Generation of Petri Nets. The generation of a Petri net representing a law is divided in three stages :

- The translation of the law in a logic expression L, by using the table 1, in order to point out a set of elementary logic expressions, $\{l_1, ..., l_n\}$.
- The deduction of a set of Petri net, $\{p_1, ..., p_n\}$ representative of each expression in $\{l_1, ..., l_n\}$, by using the table 2.
- The fusion of all the nets in $\{p_1, ..., p_n\}$ from the relations between $l_1, ..., l_n$ expressed in L, by using the table 2 to obtain a final Petri net, P, representing the law.

The final Petri net P, is embedded into the control part of each agent submitted to the law. This Petri net includes two parts: the **conditional part** with states and transitions associated with the events and states described in the APC of the law; the **deontic part** with states and transitions associated with the events or states described in the DA of the law. For example, from the law L1:

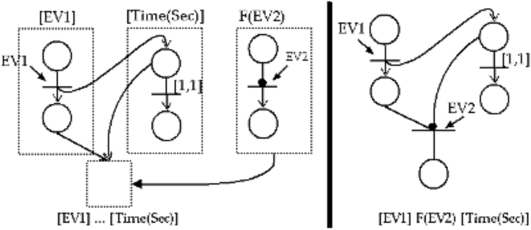

Fig. 8. The generation of the Petri net for L1

(L1) (agA : Agent and agA.type = A)(agB : Agent and agB.type= B)
FORBIDDEN(agA **do** SendingMessage **and** receiver = agB)
AFTER(agA **do** SendingMessage **and** receiver = agB)
BEFORE(1).

We can deduce the following logic expression:

$$[SendingMessage(agA, agB, M1) \wedge agent(agA, A) \wedge agent(agB, B)]$$
$$F(SendingMessage(agA, agB, M2) \wedge agent(agA, A) \wedge agent(agB, B))[time(1)]$$

From this expression, the generator deduces the Petri net representing the law. To represent a prohibition we use an inhibitor hyperarc:

Inhibitor hyperarc: *A branch inhibitor hyperarc between places $P_1, ..., P_k$ and a transition T, means that T is not firable if all the places are marked* [7].

To express a real time, we use a timed Petri net. The figure 8 represents the generation of a Petri net representing the law **L1** with inhibitor hyperarc and time.

4.6 Multiagent Laws

For a law applied to several agents, our aim is to distribute as much as possible the control into each agent affected by the law. We would like to avoid a centralized solution. So the Petri net representing the "multiagent law" is deduces as in a single agent context. Then, the net is distributed into the control parts of the agents concerned by the law. For example, the figure 9 represents the distribution of the Petri net representing the multiagent law L2 :

(L2) (agB : Agent **and** agB.type = B)(agC : Agent **and** agC.type = C)
FORBIDDEN(agB **do** CreationAgent **and** created.type = C)
IF(agC **be** State and value = available).

Let us note that the control parts of each agents 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).

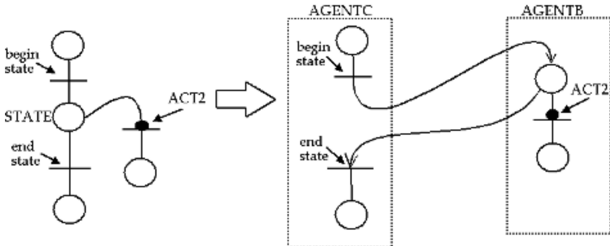

Fig. 9. Distribution between two agents

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 behavior 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 verifies the respect of its part of the law. 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.

5 Related Work

M.S. Feather *and al.* [5] treat also the agreement between a system and its requirements. In their approach, an external monitor collects events sent by the agents and a reconciler is going, when a requirement violation is detected, not to hand the system in a state that respected requirements, but to modify requirements so that they are in agreement with the new behavior. The authors do not consider essential requirements for the system execution, they do not seek to prevent inconsistent behavior. They try that a system and its requirements adapt themselves to stay in agreement during the system execution.

D.N. Lam and K.S. Barber [8] propose a methodology, the Tracing Method, to test and explain the agents behavior. The aim of this method is to ensure that an agent performs actions for the right reasons, and if an unexpected action occurred, to help explain why an agent decided to perform the action. We have in common an agent ontology to compare specifications (state-chart diagrams, communication protocol diagrams) and agents real behavior. But in our approach we propose an automation of the code instrumentation and the detection of inconsistencies between the expected and the observed behaviors. Finally, our control is embedded into agents to allow an on-line detection of errors. The Tracing Method allows an off-line analysis of the program traces generated during the system execution.

Finally, we cite the recent work of R. Paes [14]. In the context of open multi-agent systems, the authors propose the use of laws to control the emergence of wrong behaviors. If the idea is similar, the authors apply their control only to the messages passing between the agents and not to the whole behavior. They propose the use of a mediator which receives the messages, applies the laws on these messages and forwards them to the addressed agent. Here, it is about the surveillance of the agents interaction thanks to an external entity.

6 Conclusion

We have presented in this paper a framework, SCAAR, allowing the generation of agents being able to verify their own behavior. This verification consists in making sure that a set of laws associated with an agent is respected throughout the MAS execution. These laws represent requirements about agents behavior and state. The interest of our approach is principally to permit the description of laws by someone not involved in the MAS development. Another important point lies in the fact that the control can be applied to agents implemented with different kinds of agent model, in condition that the model used can be described from our agent concepts. With our framework, we provide a language to describe laws. We propose a mechanism for automatic generation of Petri nets representing the laws and insertion of control points to detect expected events. The Petri nets are used to monitor the agent behavior and detect when laws are transgressed, by using the observer approach. Finally, we propose a first solution for the enforcement of laws at the multiagent level.

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*, pages 8–15, May 1999.
2. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.
3. 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.
4. 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.
5. M.S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling System Requirements and Runtime Behavior. In *Proceedings of IWSSD9*, Isobe, Japan, 1998.
6. Y. Huang and C. Kintala. Software fault tolerance in the application layer. In *Software Fault Tolerance*, 1995.
7. R. Janicki and M. Koutny. On causality semantics of nets with priorities. *Fundamenta Informaticae*, (38):223–255, 1999.
8. D.N. Lam and K.S. Barber. Debugging agent behavior in an implemented agent system. In *Proceedings of PROMAS'04*, pages 45–56, New York City, July 20 2004.
9. Y. Liao and D. Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Trans. Software Engineering*, 18(11), November 1992.

10. J.E. Lumpp, T.L. Casavant, H.J. Siegle, and D.C. Marinescu. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. In *Proceedings of the 10th International Conference on Distributed Systems*, pages 476–483, June 1990.
11. 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.
12. M. Mansouri-Samani. *Monitoring of Distributed Sytems.* PhD thesis, University of London, London, UK, 1995.
13. 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), 1988.
14. R. Paes, G. Carvalho, C. Lucena, P. Alencar, H. Almeida, and V. Silva. Specifying laws in open multi-agent systems. In *ANIREM*, Utrecht, July 2005.
15. J. Vázquez-Salceda, H. Aldewerld, and F. Dignum. Implementing norms in multi-agent systems. In *Proceedings of MATES'04*, Erfurt, Germany, September, 29–30 2004.
16. G.H. von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.
17. Dean Wampler. The future of aspect oriented programming, 2003. White Paper, available on *http://www.aspectprogramming.com.*