Automatic Generation of Self-controlled Autonomous Agents

Caroline Chopinaud
Thales Airborne Systems & LIP6
2 avenue Gay Lussac 78851 Elancourt, France
caroline.chopinaud@fr.thalesgroup.com

Amal El Fallah Seghrouchni
LIP6
8 rue du Capitaine Scott 75015 Paris, France
amal.elfallah@lip6.fr

Patrick Taillibert
Thales Airborne Systems
patrick.taillibert@fr.thalesgroup.com

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 static techniques to validate a system does not guarantee it against the occurrence of errors in certain configurations. So, we propose an additional approach of dynamic self-monitoring and self-regulation in order that an agent might control, in real condition, its own behavior. This paper presents our approach of agent control and the generation of self-controlled agents using the observer technique in order to verify that their behavior respect a set of laws throughout their execution.

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 other agents cannot be predicted with certainty. This perspective brings up the problem of the confidence that we can have in a system execution. When critical applications are concerned, the use of such system might raise objections because of this unpredictability. So, it is essential to ensure that a multiagent system (MAS) and its agents respect some behavioral requirements, which are important for the application progress.

The aim of our research is to ensure that a MAS behavior will fulfill these requirements. A first approach should be to use classical methods, such as tests, Model Checking [2] and automatic demonstration to validate a multiagent system. But, these techniques are not often in the position to detect all possible errors and neither necessarily suitable for the validation of MAS as a whole. Even if these approaches are useful at some levels of MAS design, they let situations in which errors may occur at runtime. The Model Checking, working in a system abstraction, can only detect errors in this model; the automatic detection is complex and heavy; testing is by nature incomplete. Hence, to detect remaining errors, it is interesting to make an on-line verification of the system behavior. This verification[1] consists in monitoring and regulating inconsistent behavior to prevent a potential system failure. Moreover, we think that the agents are suitable for making 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 be in a right state once again. The control of the agents behavior will be made by the agents themselves.

Although it is possible for a developer to insert the control code into the agents, for monitoring and error detection, it can become complex: a manual instrumentation of a system program, to insert probes, is hard, time-consuming and prone to error [13]. 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) developer uses a metalanguage [11] or a li-

1 We will use the term of "control" to refer to this verification

brary of routines [8] allowing to insert probes in a transparent way, (2) the insertion will be made by compiler from the specification of the interesting events [10]. We focus on this last form and we propose a generator which creates agents being able to control themselves, from a description of requirements associated with the agents and their behavior program. In section 2 we will present our approach of agent control. In section 3, we will describe each stage of control application: the system description, the laws description and the automatic generation of self-controlled agents.

2. Control of autonomous agent

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 agents behavior. In general, norms [14] define constraints on agent 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 during the decision process (i.e. agents can act as they want and our approach consists in verifying afterwards if the chosen action respects the laws) because we want to distinguish the agent implementation and the laws/control description (see section 2.2). The laws represents significant or critical requirements defined for the system. So 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 form repairing information.

In order that the agents 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, or the agents, have a representation of this laws that they attempt to respect. 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 executed in MAS context.

2.2. Level of laws description

We wish that the person who defines the laws doesn't be necesseraly the developper. In general customers define requirements and developpers implement the system from these requirements. Also, the customer should be able to know what is important to verify whithout knowing the agents implementation. So, we suppose that laws are expressed in natural language by the customer and translated by an expert in a description language allowing the deduction of the appropriate control. Therefore, it is necessary to express the laws at a level of abstraction 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 laws description must permit to include several kinds of agent model. So the laws must state general **concepts**, representing agent models and the application. The model designer provides a set of concepts representing the model specificities and the system designer/developper provides application typical concepts. From this concepts the expert can describe the laws expressed by the customer by using the description language.

2.3. Control Enforcement

The control enforcement is ensured through five steps :

1. The model designer provides a description of concepts and their links with the model implementation.
2. The customer provide the set of laws.
3. The system developer implements the agents from a set of requirements or/and the agents use the requirements to deduce their behavior at runtime. He provides the agents regulation behavior associated with each law. He describes the concepts representing the application.
4. An expert translates the laws (2) by using a description language. The laws state concepts described in (1) and (3).
5. The automatic generation of self-controlled agents from the laws description (4), the concepts used in laws and their representation into the model implementation (1) and the code of the agent behavior (3).

3. Generation of self-controlled agents

3.1. System description

We saw that the laws are based on high level concepts allowing the description of agents and system specificities. So, we construct a basic ontology that the model and system designers will use to describe the model and the agents. A set of agent concepts is provided by D.N. Lam and K.S. Barber [9] for agent verification: *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 [5], CLAIM [6], personal agent models):*Agent*, *Object*, *Knowledge*, *Plan*, *Agent Creation*, *Message Sending*,

Message Receipt, *Migration*. Each concept has a set of attributes and methods allowing to express some tests on the concepts used in laws. The ontology can be extended to describe more precisely the models and the application by subconcepts or instances. The concepts are distributed in three categories : **Agent**, **State** (agent internal characteristics) and **Action**.

The laws stating abstract concepts, it is necessary to know their specification into the agent program to check if the laws are respected. The model designer describes the links between the concepts and the model implementation. This description can be validated to ensure that the laws will be taken into account.

3.2. Laws description

We can distinguish two kinds of law: *(1)* one representing unwanted state or behavior of an agent. It allows the detection of situations where an event which must never occurs, happens. *(2)* one representing expected state or behavior of an agent. It allows the detection of situations where an expected event does not occur. To represent this two kinds of law we use the deontic operators of prohibition and obligation. We propose a language allowing the expression of prohibition, obligation, time notion or order between events. An event can be a change in state or the execution of an action by an agent (we retrieve our three concepts categories). The semantic of our language is the standard deontic logic [15]. We divide a law in three parts :

INVOLVED_AGENTS (IA): the statement of agents involved by the law. Those are agents that can be subject to the law and agents used to describe the law application context.

DEONTIC_ASSERTION (DA): describes what is obligatory and what is forbidden. It is a set of association between an agent and an event. An association can have conditions.

APPLICATION_CONDITIONS (APC): describes conditions about law context. It is an expression describing when the DA must be respected relatively to a set of events or time.

3.3. The automatic generation

We propose to generate, from the description of the system and the laws and the behavior programs, agents being able to monitor and regulate their own behavior. We provide the agents an introspective architecture allowing self-control by adding the next components in the agent program: *(1)* some control points allowing the detection of particular events about the behavior or the state of the agent during its execution; *(2)* a control part which receives information about the detection made by the control points, analyzes the agent behavior and verifies that it respects the laws. To insert the control points into the agent we propose to use the **weaving** and to monitor and check the respect of laws we propose to use the **observer** approach.

Weaving is an important part of aspect programming. 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 classes execution after compilation. An aspect is a module representing crosscutting concerns [16]. The aspect programming, with the use of AspectC++, is shown as particularly interesting to integrate monitoring in an application [12]. We take this idea of weaving to make the automatic instrumentation of control points in the agent behavior program.

Observer [4] is an approach which consists in having a program and its model running in parallel and comparing them during the execution. The model is linked to the program by control points. Properties about the program execution are modeled, for instance, by Petri net, with transitions linked to the program by control points. When the program arrives on this control points, a controller make sure that the tokens at the right places at the right time and getting them evolve consequently. We use the observer concept to check the laws. So, each law is represented by a Petri net with transitions linked to the control points in the agent program. The control part verifies that the agent behavior corresponds actually to the model of the associated laws. When a transition is activated at a bad instant, an exception associated with the involved transition is thrown and analyzed by the treatment block of the control part to execute the repairing actions.

The generation of the Petri net representing the law and the deduction of control points is divided in two stages: *(1)* from the law, by using the language semantic and the deontic logic axiomatic, a deontic expression is deduced; *(2)* from this expression, a set of Petri nets is generated and the final Petri net representing the law is obtained by the merging of all the nets. The deontic expression put the events to observe forward and its structure allows the deduction of the final Petri net.

A Petri net representing a law is not activated over the whole execution. In fact, while the first transition is not activated the events coming from the control points in the behavior part are not treated because the prohibition and the obligation have an interest only under certain conditions. So, the law is deactivated when the net is in a deactivation state. If a law was activated all the time, off-context exception would necessarily occur, even if the behavior is correct. A solution should have been to deduce a more complete Petri net including transition and state to not generate off-context exceptions, but this deduction becomes quickly complex from laws that we want to be simple.

4. Related Works

M.S. Feather *and al.* [7] treat also the agreement between a system and its requirements. In their approach, an external monitor collects the 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 to make a system and his requirements adapt themselves to stay in agreement during the system execution.

D.N. Lam and K.S. Barber [9] 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 observed behavior. Finally, our control is embedded into agents to allows an on-line detection of errors. The Tracing Method allows an off-line analysis of the program traces generated during the system execution.

5. Conclusion

We have presented our approach of autonomous agent control, particularly mechanisms to generate automatically agents being able to check their own behavior. We use laws, describing ideal behavior or situation, that the agents must respect throughout their execution. The interest of our approach is 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 observers embedded into the agents. The next step of our works will concern the application of laws at multiagent level and the implementation of this approach into a framework : SCAAR.

References

[1] K. Barber and C. 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. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[3] M. de Sousa Dias and D. 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. dInverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dmars. Technical note 72, Australian Artificial Intelligence Institute, Carlton, Victoria, 1997.

[6] A. El Fallah Seghrouchni and A. Suna. Claim: A computational language for autonomous, intelligent and mobile agents. *LNAI*, 3067:90–110, 2004.

[7] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling System Requirements and Runtime Behavior. In *Proceedings of IWSSD9*, Isobe, Japan, 1998.

[8] Y. Huang and C. Kintala. Software fault tolerance in the application layer. In *Software Fault Tolerance*, 1995.

[9] D. Lam and K. Barber. Debugging agent behavior in an implemented agent system. In *Proceedings of PROMAS'04*, pages 45–56, New York City, July 20 2004.

[10] Y. Liao and D. Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Trans. Software Engineering*, 18(11), November 1992.

[11] J. Lumpp, T. Casavant, H. Siegle, and D. 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.

[12] 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.

[13] M. Mansouri-Samani. *Monitoring of Distributed Sytems*. PhD thesis, University of London, London, UK, 1995.

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

[15] G. von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.

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