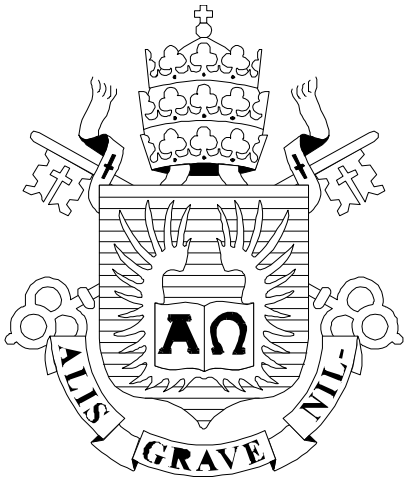

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 02/06

Treating Fault-tolerant Concerns in Models and Architectures for Multi-agent Systems

Arndt von Staa Carlos José Pereira de Lucena
Gustavo Robichez de Carvalho Jean-Pierre Briot
Ricardo Choren Zahia Guessoum

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP 22453-900
RIO DE JANEIRO – BRASIL

Monografias em Ciência da Computação, No. 02/06 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena January, 2006

Treating Fault-tolerant Concerns in Models and Architectures for Multi-agent Systems *

Arndt von Staa Carlos José Pereira de Lucena
Gustavo Robichez de Carvalho Jean-Pierre Briot Ricardo Choren
Zahia Guessoum

arndt@inf.puc-rio.br, lucena@inf.puc-rio.br, guga@les.inf.puc-rio.br
jean-pierre.briot@lip6.fr, choren@de9.ime.eb.br, zahia.guessoum@lip6.fr

Abstract: In this paper, we propose the introduction of law-governance to describe fault tolerance aspects in multi-agent systems (MAS) design models since laws delimit the things that can happen in a system. In the design-time phase, we propose an extension to the ANote modeling language to incorporate the law-governed approach. In the run-time stage, agent architectures should provide the means for the correct execution of the system, i.e. they should ensure the system will execute according to its requirements. Thus we propose the use of the DimaX. Using DimaX, we will seek to provide mechanisms to detect and recover from failures in the system. The proposed approach is detailed in the paper.

Keywords: Fault tolerance, Fault Prevention, Multi-Agent Systems, ANote, DimaX.

Resumo: Neste trabalho, nós propomos a introdução de governância para se descrever características de tolerância a falhas em modelos de sistemas multi-agentes (SMA) uma vez que leis limitam o que pode ocorrer em um sistema. Na fase de projeto, nós propomos uma extensão para a linguagem de modelagem Anote a fim de incorporar a abordagem de governância. Na fase de execução, é esperado que as arquiteturas de agentes oferecem os meios para a correta execução do sistema, i.e. elas devem assegurar que o sistema executará de acordo com seus requisitos. Assim, nós propomos o uso de DimaX. Com DimaX, esperamos oferecer os mecanismos para a detecção e a recuperação de falhas no sistema. A abordagem é detalhada no artigo.

Palavras-chave: Tolerância a Falhas, Prevenção de Falhas, Sistemas Multi-Agentes, ANote, DimaX.

---

* This work has been partially sponsored by CAPES, through the CAPES/Cofecub Program, Project EMACA number 482/05.

In charge for publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br

1 Introduction

Computer-based solutions have become distributed, increasing their scope, complexity, and pervasiveness. Agent-based computing is rapidly emerging as a powerful technology for the development of this kind of software systems. In a multi-agent system (MAS) there are several software agents, each one with its execution thread, trying to achieve its own goals. In this context, safe and reliable software operation is a significant requirement.

The dependability of a computing system is its ability to deliver service that can justifiably be trusted [Avizienis et al., 2001]. Increasing the dependability of MAS presents some unique challenges when compared to traditional systems. In MAS, agents are autonomous; they concurrently interact following local protocols; they can move etc., which makes it more difficult to verify if an agent has failed.

Modeling and implementing MAS are important open problems despite the attention they have attracted recently [Bellifemine et al. ,1999; Padgham and Winikoff, 2002; Silva and Lucena, 2004; Wooldridge et al., 2000]. In this paper, we study fault-tolerance in MAS and we identify the need to deal with it in two different levels: design level and execution level. As our primary contribution in this paper, we present an approach that separates and represents fault-tolerance concerns in MAS both in design-time models and run-time architectures.

In design-time models, we can specify rules or contracts to delimit the extension of expected behavior in the system. For instance, a specified boundary can be used to treat exceptional situations, which are not part of the system requirements but are related to the system expected execution, such as timing check faults or malicious agents. Thus, we propose the introduction of law-governance to describe fault tolerance aspects in MAS design models since laws delimit the things that can happen in a system; they are restrictions imposed by the environment to tame uncertainty and promote system dependability [Minsky and Ungureanu, 2000]. Moreover, we propose an extension to the ANote modeling language [Choren and Lucena, 2005a; Choren and Lucena, 2005b] to incorporate the law-governed approach [Minsky and Ungureanu, 1997]. ANote is a modeling language founded on agency concepts, which are described in a conceptual metamodel. ANote offers a set of diagrams to model different views of a multi-agent system.

From the run-time perspective, agent architectures should provide the means for the correct execution of the system, i.e. they should ensure the system will execute according to its requirements. The architecture can offer fault-tolerant mechanisms to deal with exceptional situations, which prevent the system to deliver its services, such as unavailability and hardware faults. We propose the use of the DimaX [Faci et al., 2004; Guessoum et al., 2005]. DimaX is the result of an integration of the DIMA multi-agent platform [Guessoum and Briot, 1999] and the DarX replication framework [Bertier et al. 2002], and it implements an adaptive replication mechanism. Using DimaX, we will seek to provide mechanisms to detect and recover from failures in the system.

1.1 Some Concepts on Fault-Tolerance

A number of terms have been used in the literature to describe dependability terms, including fault, failure and error. For the purposes of this paper, we state some definitions here. A fault is an abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function, ie a requirements, design, or implementation flaw or deviation from a desired or intended state [Leveson, 1995]. A fault is the root cause of a failure, meaning that an error is the symptom of a

fault. A failure is the inability of a system to perform its required functions within specified performance requirements [IEEE, 1990]. It occurs when an actual running system deviates from this specified behavior [Selic, 2006]. An error is a discrepancy between a computed, observed or measured value (or condition) and the true, specified or theoretically correct value or condition [IEEE, 1990]. It is the cause of a failure and it represents an invalid system state, which is not allowed by the system behavior specification [Selic, 2006].

1.2 Paper Structure

The organization of this paper is as follows. Section 2 presents two strategies that prescribe how a multi-agent system can achieve dependability. In section 3, we present our proposal to the development of multi-agent systems, which encompasses both design and architectural level concerns. Section 4 briefly describes a case study using our approach. Related work is shown in section 5 and, finally, section 6 presents some conclusions.

2 Achieving Fault-Tolerant Multi-Agent Systems

In MAS, every software agent assumes the responsibility to provide some functionality, making the system more complex. The cost and the consequences of crash faults in a MAS can be catastrophic.

The means to achieve dependability falls into two major groups [Pullum, 2001]: those employed during the software construction phase (fault prevention and fault tolerance); and those that contribute to the validation of the software after it is developed (fault removal and fault forecasting). This work is concerned with the first group. Shortly, fault prevention and fault tolerance techniques are [Pullum, 2001]:

- *Fault prevention*: to avoid or prevent the fault introduction and occurrence;
- *Fault tolerance*: to provide service complying with the specification in spite of the faults.

Fault prevention techniques are intended to keep faults out of the system at the design stage [Storey, 1996]. These techniques are employed during the software development to reduce the number of faults introduced during construction. They address, for example, system requirements and specifications, software design methods, reusability or formal methods [Pullum, 2001]. In the case of multi-agent systems in particular, the system specification plays a major role in the development of dependable solutions. We argue that not only the system specification should be as correct as possible (we understand it is an imperfect process per se), but also that the system specification documents should have additional specific information to model contexts that could lead to abnormal behavior.

Fault tolerance techniques are employed during the system development to enable software to tolerate faults remaining in the system after its deployment; and they encompass a set of activities whose goal is to remove errors and their effects from the computational state before a failure occurs. Error detection is part of the fault tolerance process and it refers to the mechanisms that identify an invalid or erroneous system state.

Some common techniques for error detection include: replication checks; timing checks; and run-time constraint checks. In the case of replication checks, multiple replicas of a software component perform the same service simultaneously. The outputs of the replicas are compared, and any discrepancy is an indication of an error in one or more of these components. In a timing check, typically a timer is started and set to ex-

pire at a point at which a given service is expected to be complete. If the service terminates successfully before the timer expires, the timer is cancelled. However, if the timer times out, then a timing error has occurred. In a run-time constraint check, some variables are checked in run-time to verify if they do not exceed their boundary value.

3 Developing Fault-Tolerant Multi-Agent Systems

In MAS, software agents may be independently implemented, ie the development may be done without a centralized control. In order to have a coherent dependable system, it is important to use fault avoidance and fault tolerance mechanisms during the system specification and implementation.

3.1 Fault Prevention Specification

As we mentioned before, software specification is a rather imprecise activity. Imprecision contributes to the development of less dependable systems. Fault prevention is especially useful when it is possible to predict the nature of failures which can occur. In MAS, fault prevention is best suited to those situations which are simple (to describe) but the techniques are still valuable. Thus all feasible fault prevention techniques should be used during the design of the system to minimize the failures that can occur.

Some categories of failures, mainly those related to the system expected behavior, can be foreseen. The moment we incorporate them in models, architectures or high level specifications, they stop being failures in absolute terms, and start being part of the system requirements. In other words, if a fault that provokes such a failure is exercised, some handler will take care of the now foreseen failure. In many cases we are even able to run tests to verify whether the failure is adequately taken care of (handled).

Possibly the modeling language used will need some additional construct to allow specifying the behavior and the point of capture of the now possibly tamed failure. This is the underlying idea of extending ANote to specify the environment and its laws.

3.1.1 The Extended ANote Modeling Language

ANote has a conceptual metamodel that defines a schema of the concepts, their relations, and some constraints that build a multi-agent solution. It provides seven views (each one based on a metamodel concept) which specify the system's goals, agents, organizations, scenarios, plans, interactions and resources (for further details about ANote, refer to [Choren and Lucena, 2005a; Choren and Lucena, 2005b]).

This work introduces law as another ANote metamodel concept. In fact, a law is a specification of a constraint and, as such, should be part of the system models. It defines the circumstance that triggers a situation and what should be done next. Note that it may not have knowledge about the cause of the situation. The law governed approach is suitable to tame this uncertainty since it can be used to create a boundary of tolerated behavior in the system.

As the metamodel concepts guide the other ANote views, the introduction of the law concept into the ANote metamodel will generate a new view, with a new notation, to the language. The regulatory view captures the system law, and the main model element in this view is the law. Each law is treated as a contract, which, when broken, shall give evidences of system faults.

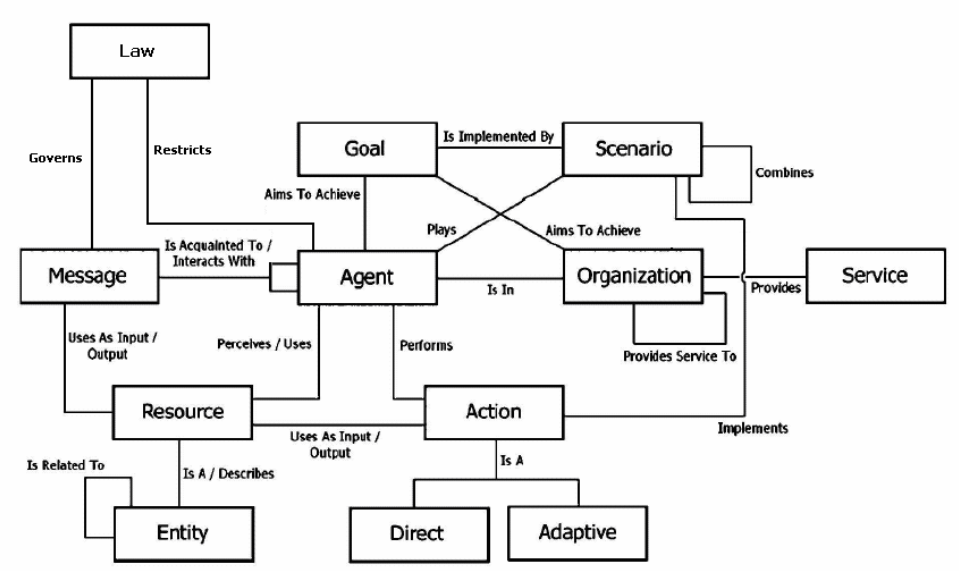

Figure 1. Extended ANote metamodel

In this new ANote view, laws are represented as text in XMLaw description language [Paes et al., 2005]. XMLaw illustrates the structure and the relationship of law elements, very similar to a contract specification. It uses the abstraction of scenes, which are composed of protocols, clocks, actions and norms.

For example, let's consider the payment protocol between assembler and bank agents. Basically, the payment is made through a payment message sent by the assembler to the bank and the bank reply with the confirmation response represented by the receipt message. Suppose the bank should send the reply message within five seconds. The XMLaw specification of this rule is the following:

```
<Transition id="payTransition" from="p1" to="p2" message-ref="payment">
  <ActiveNorms>
   <Norm ref="PermissionToPay"/>
  </ActiveNorms>
 </Transition>
...
 <Clock id="timeout" type="regular" tick-period="5000">
  <Activations>
   <Element ref="orderTransition" event-type="transition_activation"/>
  </Activations>
 </Clock>
...
 <Permission id="PermissionToPay">
  <Owner>Assembler</Owner>
  <Activations>
   <Element ref="orderTransition" event-type="transition_activation"/>
  </Activations>
  <Deactivations>
   <Element ref="timeout" event-type="clock_tick"/>
  </Deactivations>
 </Permission>
```

This law can be used to specify a possible fault circumstance in the system, more specifically a timing check fault. If the bank does not respond in five seconds, the environment can signal that, possibly, the bank agent is in failure. The law could be written so that the system signals that an alternate behavior should be done. For in-

stance, it could tell the assembler to try other 5 times and if there is still no answer, the system would finally signal the fault.

It is important to mention that, similarly to the other ANote's seven views, this new view shows a particular perspective of the system, which is related to the others. This makes it possible to create consistency checks of the system diagrams. In the example shown above, the law diagram is directly connected to an interaction diagram that specifies the message exchange between the assembler and bank agents.

3.2 Fault Tolerant Architecture

Despite all our efforts, programs will contain faults. Some of these faults may even be external to the system at hand, for example operating system faults, surges in energy and hardware faults. Thus, some faults will remain causing unforeseen failures, some of which will never be exercised.

In MAS, it is important for the run-time architecture to be able to keep the system running despite some errors. This work uses DimaX [Faci et al., 2004; Guessoum et al., 2005] as a fault tolerant execution environment. DimaX is a fault-preventive architecture that uses replicas to replace failed components.

3.2.1 Error Detection

The essence of error detection is monitoring, ie the agents must be observed somehow to detect time and behavior failures. Monitoring gives information to decide whether an agent output (or lack of it) is incorrect. Two things can then be done: measuring the passage of time and exploring replication.

DimaX provides MAS with services such as distribution, replication, monitoring and naming service, and fault detection [Marin et al., 2003]. Two replication strategies (active and passive) can be used to replicate agents. Active replication provides a fast recovery delay, being more suitable for applications with real-time constraints. Passive replication provides a low overhead under failure but it does not provide short recovery delays. So, the choice of the most suitable strategy relies on the environment context. Active replication must be chosen when the failure rate becomes too high or when the application has real-time constraints; otherwise, passive replication is most suitable.

In most MAS applications, the environment context is very dynamic. So, the choice of the replication strategy of each component, which relies on a part of this environment, must be determined dynamically and adapted to the environment changes. Moreover, a MAS component, which can be very critical at a moment, can loose its criticality later. If we consider the replication cost to be very high, the number of replicas of these components must be therefore dynamically updated. Thus, the solution proposed by DimaX allows the dynamic adaptation of the number of replicas and of the replication strategy. This solution is provided by the framework DarX.

DarX provides the needed adaptive mechanisms to replicate agents and to modify the replication strategy. Meanwhile, we cannot always replicate all the agents of the system because the available resources are usually limited. In large dynamic applications, the roles and relative importance of the agents can vary during the system execution. For example, agents are able to change roles, engage in new interactions and initiate new forms of cooperation. Also, new agents may also join or leave the application (open system), thus it is important to have a mechanism to increase the system reliability [Bertier et al., 2003].

DimaX allows the MAS to dynamically identify the most critical agents and to decide which reliability strategy to apply. A MAS is thus automatically augmented

with a set of monitor agent to use various levels of information such as communication load (system level) and roles (application/agent level) to define the most critical agents and to replicate them accordingly.

3.2.2 Error Recovery

To recover from an error, the system needs to be restored to a valid state. This recovery requires reconfiguration, renegotiation between interacting parties, rebinding, resynchronization, and so on. In DimaX, error recovery is directly related to the use of replicas. If an agent is in a fault state and it has a replica, the architecture dynamically changes the agent for its replica so that the systems can continue its normal execution.

If an agent is not replicated, then this is a problem of diagnostic check, which is typically related to component auditing to check its normal execution. Diagnostic checks are out of the scope of DimaX.

4 Case Study

In this section we present a proof of concept example, based on the Expert Committee agent system. The Expert Committee defines a simple set of requirements for a peer-review system. This application is composed of planning and coordinating activities related to paper submission, distribution, review and notification. These activities involve different participants, and their coordination is very important.

In this example, we chose the scenario of the paper distribution activity between the chair and reviewer agent to explain our proposal. This scenario involves the chair agent proposing a paper to the reviewer. The reviewer has to inform if it accepts the paper to review and, finally, it reviews the paper (if accepted with success). The ANote agent and interaction view diagrams that specify this scenario are the following (figs 2 and 3):

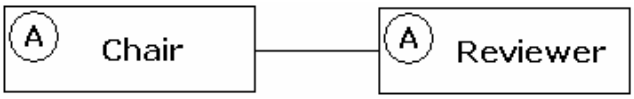

Figure 2. Agent view diagram for the Expert Committee scenario

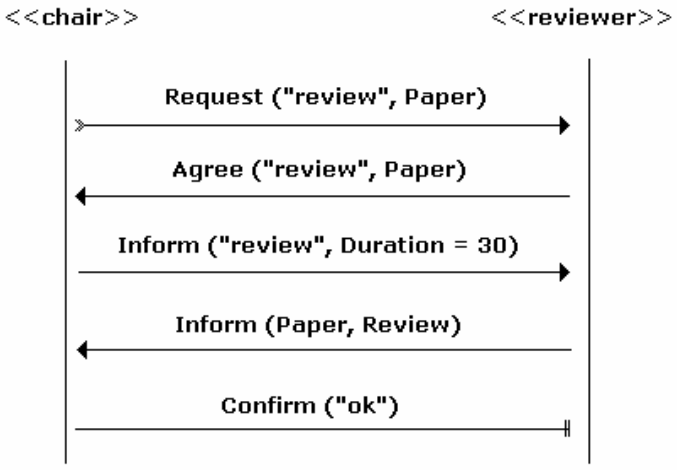

Figure 3. Interaction view diagram for the Expert Committee scenario

4.1 Introducing the Fault Prevention Specification

The Expert Committee specification does not provide fault prevention information. In this scenario, we will introduce the following restrictions:

Upon receiving a request to review a paper, the reviewer has to agree to review it (ie send an Agree message) between 15 to 30 minutes after receiving the request. The chair then informs that the review has 30 minutes to review the paper.

These new information is not directly pictured in the ANote diagrams shown above. In fact, this negotiation between chair and review can be used to define a scene where environmental laws will handle them. The XMLaw specification of this improved interaction is listed below:

```
<Clock id="timeToStart" type="regular" tick-period="1800000">
  <Activations>
   <Element ref="requestReviewersTransition"
            event-type="transition_activation"/>
  </Activations>
 </Clock>

 <Clock id="timeOut" type="regular" tick-period="3600000">
  <Activations>
   <Element ref="requestReviewersTransition"
            event-type="transition_activation"/>
  </Activations>
 </Clock>

 <Permission id="PermissionToSend">
  <Owner>Reviewer</Owner>
   <Activations>
    <Element ref="timeToStart" event-type="clock_tick"/>
   </Activations>
    <Deactivations>
     <Element ref="timeOut" event-type="clock_tick"/>
   </Deactivations>
 </Permission>
 <Transition id="sendReviewTransition" from="r2" to="r3"
             message-ref="review">
  <ActiveNorms>
   <Norm ref="PermissionToSend"/>
  </ActiveNorms>
 </Transition>
```

Note that, with this law, we provide mechanisms to handle fault prevention for two types of faults: timing and malicious agents. It can also be used to state unavailability fault, but this will be directly handled by the run-time architecture, as we will show later. The timing fault we try to prevent with this specification happens twice: on the deadline on the agreement and on the review sending. If the reviewer agent does not comply with the deadlines, it may indicate that it is in a failure state.

The malicious agent fault the specification intends to prevent is the following. Imagine there is a malicious agent listening to the conversations that take place in this system and that this agent wants to pass as a reviewer. He then can listen to the request message and, unaware of the 15 minutes delay law, he sends an agreement to the chair in order to make believe that he is a reviewer. However, the law mechanism will prevent this agent to pass as the reviewer in this case.

4.2 Introducing the Fault Tolerant Architecture

In DimaX, a multi-agent system is automatically enriched with a set of monitoring agents that observe and control the system. The monitor agents' behavior [Guessoum et al., 2005] relies on an adaptive replication mechanism. These monitor agents work with a parameter set. In our case, these parameters come from the laws and other constraint defined in the extended ANote models. The monitor agent thus observes the interaction (ANote interaction view diagram) and the behavior (ANote planning view diagram) of the domain agents (ANote agent view diagram) and uses the provided laws (ANote law view diagram) to detect the fault.

In the Expert Committee system, there are two types of monitors: chair and reviewer monitors. The chair monitor is responsible to monitor the chair agent and to replicate it in order to avoid its failure. Besides, the monitor observes the interactions. For instance, after each request, the monitor initiates a clock (according to the law specification). Upon receiving a response, it is able to check if the sender is a reviewer and if the response was sent between 15 to 30 minutes. Otherwise, the monitor agent deletes the message and its sender is memorized as a malicious agent.

Thus, monitoring is done using the information available from the design-time models. In case of crash faults, such as those mentioned in section 3.2, the architecture uses replicas to keep the system up. Replication is done transparently by DimaX.

5 Related Work

Several works have underlined the importance of fault-tolerance in multi-agent systems. Many approaches have thus been proposed to deal with some aspects of the fault tolerance problem. We distinguish two main approaches: corrective [Fedoruk and Deters, 2002; Hagg, 1997] and preventive [Horling et al., 2002; Kaminka et al., 2002; Malone et al., 1997]. The preventive approach deals with the ability to continue to deliver services when faults occur.

Hagg [1997] introduces sentinels to protect the agents from some undesirable states. Sentinels represent the control structure of their multi-agent system. They need to build models of each agent and monitor communications in order to react to faults. Each sentinel is associated by the designer to a functionality of the multi-agent system. This sentinel handles the different agents which interact to achieve the functionality. The analysis of his beliefs on the other agents enables the sentinel to detect a fault when it occurs. Adding sentinels to multi-agent systems seems to be a good approach; however the sentinels themselves represent failure points for the multi-agent system. Moreover, the problem solving agents themselves participate in the fault-tolerance process.

Fedoruk and Deters [2002] propose to use proxies to make transparent the use of agent replication, i.e. enabling the replicas of an agent to act as a same entity regarding the other agents. The proxy manages the state of the replicas. All the external and internal communications of the group are redirected to the proxy. However this increases the workload of the proxy which is a quasi central entity. To make it reliable, they propose to build a hierarchy of proxies for each group of replicas. This approach lacks flexibility and reusability in particular concerning the replication control. The experiments have been done with FIPA-OS which does not provide any replication mechanism. The replication is therefore realized by the designer before run-time.

In the corrective approach, the process consists of fault diagnostic and repair. Kaminka et al. [2002] adapt a monitoring approach in order to detect and recover faults. They use models of relations between mental states of agents. They adopt a procedural plan-recognition based approach to identify the inconsistencies. However, the

adaptation is only structural, the relation models may change but the contents of plans are static. Their main hypothesis is that any failure comes from incompleteness of beliefs. This monitoring approach relies on agent knowledge. The design of such multi-agent systems is very complex. Moreover, the behavior of agent cannot be adaptive and the system cannot be open.

Horling et al. [2002] present a distributed system of diagnosis. The faults can directly or indirectly be observed in the form of symptoms by using a fault model. The diagnosis process modifies the relations between tasks, in order to avoid inefficiencies. The adaptation is only structural because they do not consider the internal structure of tasks. The different diagnosis subsystems perform local updates on the task model. However, performance is optimized locally but not globally.

The work of Malone et al. [1997] on coordination relies on a characterization of the dependencies between activities in terms of goals and resources. These dependencies represent situations of conflict, and the different coordination mechanisms represent the solutions to manage them. The main contribution of this approach is the proposed taxonomy of these dependencies. The authors offer a framework of coordination study which provides the basic stone to build a monitoring approach. However, this monitoring approach has not yet been developed.

These approaches present useful solutions to the problem of monitoring in multi-agent systems. However, the monitoring component is often centralized and its design relies on the agents' knowledge.

6 Summary and Future Directions

In this paper we introduced the environment as a major concept to separate and represent fault-tolerant concerns in two levels: design-time models (application environment) and run-time architectures (execution environment) for MAS. Using the environment as a guide, we showed that fault tolerance should be considered since early stages of the development and that it should not require the implementation of particular strategies for a specific application. Our contributions can be summarized in the following points:

- Application models should present fault prevention mechanisms. The law governed approach is suitable to cope with the fault prevention mechanisms since it defines boundaries of tolerated agent behavior, fostering the development of trusted systems.
- To support the specification of fault prevention mechanism in design-time models, we propose an extension to the ANote modeling language by supporting the specification of laws.
- The execution architecture should present fault tolerance mechanisms. Error detection and recovery strategies are key issues in these architectures.
- To support the implementation of fault tolerance mechanisms in run-time architectures, we propose the use of DimaX.

Apart from the theoretical interest of separating the concerns in the specification and the architecture levels, we showed a practical example of how this separation can improve fault tolerance in agent systems. While the ultimate goal of the techniques described in this paper is to build more robust MAS, there are rational steps along the way which must be developed first. For example, small scale systems, like the one presented, should be constructed first.

We also understand that the fault tolerance solution presented here is not complete. For instance, the use of replicated exact copies of software agents alone cannot increase the reliability if software faults are present. One solution would be for the sys-

tem to provide diversity in both design and implementation of software agents. The purpose of diversity is to make the agents as diverse and undependable as possible, with the goal of minimizing identical error causes [Pullum, 2001].

For the future, we believe that law may have other usages such as help type checking, and testing and formal verification to remove logic errors. As for the implementation architecture, some other aspects such as diagnostic checks and damage confinement may be implemented to increase fault tolerance.

Acknowledgements

This work is partially supported by the CAPES/Cofecub Program under the project "EMACA", number 482/05.

References

AVIZIENIS, A.; LAPRIE, J. and RANDELL, B. Fundamental Concepts of Dependability. Research Report N01145, LAAS-CNRS, 2001.

BELLIFEMINE, F.; RIMASSA, G. and POGGI, A. JADE – A FIPA-Compliant Agent Framework. In Proceedings of the Fourth International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM 1999). 1999.

BERTIER, M.; MARIN, O. and SENS, P. Implementation and Performance Evaluation of an Adaptable Failure Detector. In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002). IEEE Computer Society: Massachusetts, ISBN: 0769515975, 2002, p. 354-363.

CHOREN, R. and LUCENA, C. Modeling Multi-agent systems with ANote. In: Software and Systems Modeling 4(2), Springer, p. 199-208, 2005.

CHOREN, R. and LUCENA, C. The ANote Modeling Language for Agent-Oriented Specification. In: R. Choren, A. Garcia, C. Lucena and A. Romanovsky (eds.) Software Engineering for Multi-Agent Systems III - Research Issues and Practical Applications. Lecture Notes in Computer Science, LNCS 3390. Springer: Berlin, ISBN: 3540248439, 2005, p. 198-212.

FACI, N.; GUESSOUM, Z.; MARIN, O. and LASKRI, M.T. DimaX: A Fault-Tolerant Multi-Agent Platform. In Proceedings of the International Conference on Advances in Intelligent Systems - Theory and Applications (AISTA 2004), 2004.

FEDORUK, A. and DETERS, R. Improving fault-tolerance byreplicating agents. In Proceedings of AAMAS 2002, vol. 2, p. 737-744, 2002.

GUESSOUM, Z.; FACI, N. and BRIOT, J-P. Adaptive Replication of Large-Scale Multi-Agent Systems – Towards a Fault-Tolerant Multi-Agent Platform. In A. Garcia, R. Choren, C. Lucena, A. Romanovsky, T. Holvoet, P. Giorgini (eds.) Proceedings of the Fourth International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2005). ACM Press: New York, ISBN: 1595931163, 2005, p. 62-67.

GUESSOUM, Z. and BRIOT, J-P. From Active Objects to Autonomous Agents. IEEE Concurrency 7(3), IEEE Computer Society, p. 68–76, 1999.

HAGG, S. A sentinel approach to fault handling in multi-agent systems. In C. Zhang and D. Lukose, editors,Multi-Agent Systems, Methodologies and Applications,number 1286 in LNCS, pages 190–195. Springer Ver-lag, 1997.

HORLING, B.; LESSER, V.; VINCENT, R. and WAGNER, T. The Soft Real-Time Agent Control Architecture. In Proceedings of the AAAI/KDD/UAI-2002 Joint Workshop on Real-Time Decision Support and Diagnosis Systems, 2002.

IEEE. IEEE Standard Glossary of Software Engineering Terminology. IEEE Computer Society: Massachusetts, ISBN: 155937067X, 1990.

KAMINKA, G.A.; PYNADATH, D.V. and TAMBE, M. Monitoring teams by overhearing: A multi-agent plan-recognition approach. Journal of Intelligence ArtificialResearch 17, p. 83–135, 2002.

LEVESON, N.G. Safeware: System Safety and Computers. Addison-Wesley Professional: Massachusetts, ISBN: 0201119722, 1995.

MALONE, T.W.; LAI, K. and GRANT, K.R. Agent for Information Sharing and Coordination: AHistory and Some Reflections, Software Agents, AAAI Press, p. 109-143, 1997.

MARIN, O.; BERTIER, M. and SENS, P. DarX - A Framework for the Fault-Tolerant Support of Agent Software. In Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003) IEEE Computer Society: Massachusetts, 2003, p. 406-416.

MINSKY, N. and UNGUREANU, V. Regulated Coordination in Open Distributed Systems. In D. Garlan and D. LeMetayer (eds.) Proceedings of the Second Conference on Coordination Languages and Models. Springer: Berlin, 1997, p. 81-97.

MINSKY, N. and UNGUREANU, V. Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. ACM Transactions on Software Engineering and Methodology 9(3), ACM Press, p. 273-305, 2000.

PADGHAM, L. and WINIKOFF, M. Prometheus: A Methodology for Developing Intelligent Agents. In Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002). ACM Press: New York, 2002, p. 37-38.

PAES, R.B.; CARVALHO, G.R.; LUCENA, C.; ALENCAR, P.S.C.; ALMEIDA, H.O. and SILVA, V.T. Specifying Laws in Open Multi-Agent Systems. In Proceedings of Agents, Norms and Institutions for Regulated Multiagent Systems (ANIREM 2005), 2005.

PULLUM, L.L. Software Fault Tolerance: Techniques and Implementation. Artech House Publishers: Massachusetts, ISBN: 1-580-53137-7, 2001.

SELIC, B. Fault Tolerance Techniques for Distributed Systems. URL: http://www-128.ibm.com/developerworks/rational/library/114.html Accessed on: 01/2006

SILVA, V.T. and LUCENA, C. From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language. Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers 9(1-2), Springer, p. 145-189, 2004.

STOREY, N. Safety-Critical Computer Systems. Addison-Wesley: Harlow, ISBN: 0201427877, 1996.

WOOLDRIDGE, M.; JENNINGS; N.R. and KINNY, D. The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers 3(3), Springer, p. 285-312, 2000.