Modeling Exception Management in Multi-Agent Systems

Eric Platon

Doctor of Philosophy

Department of Informatics,
School of Multidisciplinary Sciences,
The Graduate University for Advanced Studies (SOKENDAI)

2007

January 2007

Abstract

Multi-agent systems are often presented as the next major generation of software to cope with the increasing complexity in modern applications. MAS are distributed systems of autonomous and interacting entities named agents. They are possibly large-scale systems and the agent research community aims at having agents collaborate or compete with one another to achieve their functions in a highly modular and flexible way. A variety of applications of agent technologies can be observed in state-of-the-art software developed from autonomous robots in manufacturing to software agents that assist users over the Internet. Multi-agent systems are therefore promising models and technologies in the future advances in Software engineering and Artificial intelligence.

Multi-agent systems are software in the first place, and 50 years of history in Computer science has shown that constructing dependable systems requires dedicated endeavors and practices. Dependability refers to qualities of a system, in terms of availability to the user of the system, reliability to provide the functions it is designed for, and safety and security of execution. Fault tolerance techniques were developed in traditional Software engineering to increase the degree of dependability of software, and current achievements allow guaranteeing several of the aforementioned qualities in many cases of close and homogeneous systems. Multi-agent systems challenge the current achievements and target more complex systems, as required in the current demand from software users and the infrastructure of our society. Multi-agent systems target open and heterogeneous systems of autonomous agents.

Among the techniques to increase the dependability of software systems, exception handling is notably famous for its strength and simplicity. Programming languages have for long exception handling capabilities to process conveniently and systematically exceptional conditions encountered during a program execution. Distributed computing has however shown that exception handling required specific extensions in the case of distributed applications, and work on software architectures and component-based development have shown the need for other models as well. Multi-agent systems set forth challenging properties that also need to reconsider the question of exception.

The aim of this thesis is to study the notion of exception in Multi-agent systems and to propose a framework adapted to the challenges of openness, heterogeneity, and especially the autonomy of agents. Related work in the agent community has achieved in the past a number of results that showed the need for system-level exception management in Multi-agent systems. The management encompasses handling and the required mechanisms around the handling. The achievements to date set limitations on the type of MAS they can apply to. Agents are often not autonomous and the system-level approaches require agents to perfectly collaborate in the exception management procedure. In this thesis, the ability of agents to deal with exceptions by themselves in the first place is seen as a prerequisite to guarantee autonomy. Exception management then relies on agent-level mechanisms to cope with the shortcomings of current achievements and complement them. Agents keep the capability to freely choose when to initiate exception handling, and when to accept system-level support or rely on individual skills.

The approach developed in this thesis ensures the autonomy of agents by a novel execution model that guarantees the agent preserves control of itself all along its execution and despite the occurrence of exceptions. The model lets the agent decide whether an event is an exception as an individual decision, thus enforcing further the autonomy. The model is formally described and a corresponding software architecture is proposed to implement it. The architecture is subsequently applied to a case study to validate the approach, compare

it to existing work, and evaluate its computational cost. The perspectives of this work lie in a number of challenges that can be further elaborated in the framework proposed in this thesis. In particular, the automatic generation of handling strategies by agents in a range of situations is a promising capability that can expand the autonomy of agents in dealing with various exceptional situations. Another notable research direction is the evaluation by an agent of handling strategies received from other agents in the system. The interest in this topic is particularly relevant in future endeavors to bridge previous work, that essentially provide agents with strategies, with the present approach for autonomous agents that are able to estimate when such an external support is acceptable.

Contents

List of Figures

List of Tables

Acknowledgments

The work presented in this document would have never been completed without the support of five people. Pr. Shinichi Honiden was my main support in research and more generally at work. All my thanks to him for the chance to join his laboratory and spend time under his supervision. Dr. Nicolas Sabouret was also my main support and deserves all my thanks as well, even 10,000 km away from my living place. His advice and continuous endeavors for me were always helpful. Pr. Ken Satoh played an essential role in my understanding of the research world. All my thanks to him as well for showing me that we live on models and interpretations, but research must look through them.

All my thanks go especially to the fourth and fifth persons. My wife Kumiko supported me everyday of my research in research. She took good care of me and accepted me to disappear in my world while physically present. She was also the greatest support to take care of our daughter Louca, the fifth one and source of my energy. Kumiko and Louca saw me while my mind was somewhere else, especially when completing this document, and their support and understanding was the most precious present from them to accomplish this work.

I need also thank Dr. Paul Guyot for sharing with me many discussions and thinkings about research, engineering, and other social or technical themes that we still like to discuss. Paul also played an important role in my understanding of research and our mutual understanding—and misunderstandings—have seemingly contributed to both of us. Dr. Cyrille Artho has taught me many things and I would like to thank him for all that. I will never manage `tr` as well as he does, but our common work was a great experience. Dr. José Ghislain Quenum has also taught me many things on both technical and social matters. Our discussions were great developments and considerations about topics as various as software architectures or cultural considerations. Kazutaka Matsuzaki and Fuyuki Ishikawa deserve special thanks as we went through the Ph.D. course together from the beginning to the end. Kazutaka and Fuyuki have supported me in my Japanese life all the time. I could not cope with administration matters without their help and advice. Kazutaka had to remind me the shared passwords so many times that I have to thank him for that! I would also like to thank the researcher team of the institute, who advised me so many times about the conduction of my research, but also about cultural matters. Nobukazu Yoshioka, Yasuyuki Tahara, Kenji Taguchi, and Hideaki

Takeda were strong supports along my course for the Ph.D. Although I would like to thank everyone individually in our laboratory, I simply list the names of all the persons that supported me, and accepted to interact with my broken Japanese and try hard to find what I wanted or needed. My thanks go to, in spatial distance from my desk, Kenji Tei, Makoto Ishiguro, Yuki Matsuoka, Yuichi Sei, Ryota Seike, Truong Khanh Quan, Takuo Doi, Takuya Karube, Satoshi Kataoka, Satoshi Niwa, Ryu Tatsumi, Hiroyuki Nakagawa, Eric Tschetter, Shunichiro Suenaga, our dynamic and so helpful Sayo Omata, Miki Nakagawa, and Shuko Yamada, and even farther Yasmine Charif-Djebbar and Laurent Mazuel.

My research has also developed enormously by interacting with external persons, essential ingredients in the research community metaphor. My encounter with Danny Weyns and our continuous interactions allowed to develop our ideas, notably on the question of environment for Multi-agent system that Danny led to remarkable and significant research achievements. Although my present thesis does not reflect the work done with Danny, the work has been improved from our interactions and it will continue improving further in the directions we are still discussing today. I would like to thank researchers with who I worked occasionally and who guided me directly or indirectly. All my thanks to Van Parunak, Andrea Omicini, and Marco Mamei for the advice and energy put into working together on research papers. I also learned a lot from interactions with Jeff Kramer, Robert Kowalski, Marie-Pierre Gleizes, Christophe Roche, and Eric Merle. I would like to send all my thanks to them for their direct or indirect contributions to the development of my research and the understanding of research.

The thanks are incomplete, no doubt about that. My apologies to the ones I omitted in these acknowledgments, but I keep in my head enough thanks to always remember their contribution to this work.

Eric Platon, Tokyo, November 2006.

One

Introduction

Multi-agent systems are among the latest generations of intelligent software systems. They consist of software programs named agents that execute in parallel and interact to achieve functions of the system in an environment [106, see the prologue]. They are used for example when a complex problem can be decomposed into simpler sub-problems: Agents solve assigned sub-problems and they interact to provide a global result. The particular trait of Multi-agent systems lies in the property that agents are supposed *autonomous decision making entities*. In other words, agents interact with one another to accomplish their tasks, but they have *no direct control over others* and *they can refuse to interact*.

A consequence of autonomy is that the state of agents is private and cannot be read or modified by others. Agents have then individual strategies to interact and to reveal or hide contents in their states. In that sense, the notion of autonomy is well suited to the present demand in software. The rise of the Internet and the globalization of activities tends to have *individuals interact more and more through distributed computers*, where individuals are either companies, institutions, or human users—all of them autonomous entities that wish to preserve their 'internal information'. The software industry needs to support the activities of these actors, either on-line over the Internet, or inside a smaller context such as an intranet. These actors can be thought of as agents that act autonomously in a 'social system', and that is why MAS particularly fit the current needs. MAS are appropriate, possibly distributed, software architectures to deal with these issues and provide adapted solutions to the software industry. The increasing interest in Service-oriented architectures is an indicator of this trend to have individuals interact on-line while controlling the type and amount of information they expose [94].

Beyond the adequacy of MAS to current needs in the software industry, autonomous agents are also promising approaches to the ever-increasing requests for automatic processing of tasks. The aforementioned individuals coordinate to conduct their activities and many tasks are repetitive, even redundant, but their performance requires a certain degree of autonomy. Introducing artificial agents to assist or replace individuals in the performance of these tasks has been a target

since the advent of AI with 'single-agent system' [68, 83], and MAS has opened a large number of challenges and applications. For example, medium and large teamwork requires adequate and accurate project scheduling. The introduction of a MAS to assist each team player in scheduling meetings and other shared activities can help improving team work [11].

Some other traits are also important in MAS, notably that they are *open*, *interoperable*, and *heterogeneous* systems. Openness allows agents to enter and leave the system dynamically. Interoperability refers to the existence of common coordination and interaction mechanisms, notably message-passing in the case of agents [23]. Heterogeneity means agents can rely on different architectures, programming languages, or mechanisms to take part into the system, provided they just comply with the interoperability assumption. These three properties are also particularly interesting to address the present needs of the software industry.

The challenges. Multi-agent systems appear as an adequate approach to current challenges in many areas. The current state of research and development cannot provide however certain characteristics that users and designers require from modern software systems, and that were originally promised by agent technologies as 'desirable properties' [106, p.8]. Two of the characteristics that remain difficult to achieve are *dependability* and *resilience*, both concepts related to how MAS react to unexpected situations, i.e. *exceptions*.

Dependability refers to qualities of a software system, in terms of availability, reliability, safety, and security [4]. In other words, a system is dependable if it is available when the user needs it, it can provide continuous service, it does not cause any harm, and it guarantees the privacy of individuals. In the context of MAS, much research is actually part of traditional software engineering concerns, for the major part in the Distributed Computing domain. Little work deals explicitly with issues specific to MAS [49, 39, 38]. Typically, fault-tolerance techniques such as monitoring and replication are considered in the context of autonomous agents to guarantee a level of dependability. The principal issue with the current achievements is that it is difficult to find an agent-oriented technique that provides both convenience as a software engineering approach and full respect of the properties of MAS, most notably the autonomy assumption, but also the open, interoperable, and heterogeneous characteristics.

Resilience of a system is the capability for the system to achieve its purpose despite internal problems and the immersion in dynamic and often unreliable environments. Resilience then characterizes how well a MAS adapts to internal or external stress. MAS with no resilience function improperly whenever the dynamics of their agents has unexpected fluctuations or the conditions required in the environment are not met. MAS with high resilience can conversely adapt to changes in the dynamics of their agents or the environment, and continue to function properly. Resilience is therefore a mean to achieve dependability. It is however a challenging mean as it is related to the concepts of 'self-healing software' and more gener-

ally 'autonomic computing' [3, 90]. Autonomous agents are expected to be resilient, i.e. to accomplish there activities despite sources of stress. The dependability of MAS can therefore rely on the resilience of the agents. Current issues with resilient agents are that most techniques tend to be 'macro-approaches' as they deal with the system as a whole, by opposition to a 'micro-approach' that focuses on the agent. Distributed algorithms, interaction protocols, and other 'system-level' mechanisms have been developed as macro-approaches in order to have agents execute with some forms of resilience [66, 56, 53, respectively]. However, micro-approaches have much less achievements, despite the potential of having agents really resilient with respect to their autonomy. Little work has been conducted, such as self-controlled agents and commitment protocols [13, 59, respectively], and a number of issues remain to be addressed, including the combination of macro- and micro-approaches.

The reason why macro-approaches are more developed can be explained in the perspective of engineering software and for the sake of efficiency. Macro-approaches adopt a global view on the problem at hand, divide the tasks to adapt to change (or recover from problems), and distribute them to agents. These 'descending' methods are well-known in many research area such as Management or the Manufacturing industry, so that the introduction in MAS is eased. Micro-approaches rely however on 'ascending' methods where each agent is endowed with functionalities for resilience, and the resilience in shared activities 'emerges' from the interactions. Micro-approaches are therefore more complex to engineer and control, especially for large-scale systems where the number of agents can be high. Macro- and micro-approaches are however complementary. The former is usually a service external to agents and it may fail in some circumstances. The latter can then maintain the resilience of the system owing to the resilience of each agent. In addition, most macro-approaches assume that agents are *cooperative* in the adaptation or recovery methods. This assumption is however interfering with the autonomy of agents, which rightly allows an agent to refuse a cooperation for private reasons (e.g. cooperation is too slow or too costly). Current work is therefore brittle facing such kind of decisions [53].

Toward exception management. Among the different ways to improve the dependability and resilience of MAS, a number of techniques exist in Software engineering, Distributed computing, and Artificial intelligence that have been introduced in MAS under some assumptions. *Exception handling* is one of them as it stands for many years in programming languages as a convenient and powerful technology, yet simple in its principles [34]. When a program has to process unexpected information (e.g. missing parameters, unknown format), an exception handling system (EHS) integrated in the program provides mechanisms to deviate the execution flow toward a 'handler', i.e. a piece of code tailored to handle a specific situation on behalf of the program. The EHS directs the execution flow on completion of the handler back to the program. The basic handling mechanism is illustrated on Fig. 1.1.

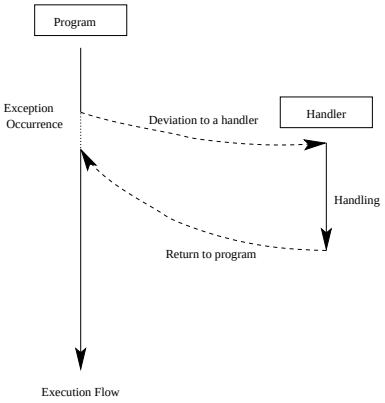


Figure 1.1: Basic exception handling in programming languages

An EHS contains in fact additional mechanisms to deal with situations such as the search for handlers along the program call-stack when no handler is available at the point where the exception was declared. The call-stack is a record of the series of operation invocations that are done in the execution of the program. If no handler is available at the point where the exception occurs, a handler is searched and asked to the previous 'caller' in the stack. The search continues until a handler is found or when the call-stack is entirely 'rewound', which means the program cannot handle the exception at all and must terminate.

In the context of MAS, the idea of having agents perform exception handling the same way as shown on Fig. 1.1 is attractive, but two challenges make difficult the use of this mechanism, namely distribution and autonomy. Research in distributed computing has shown that the basic semantics of exception handling is not sufficient to deal with problems such as *concurrent exceptions* [112, 43]. In distributed software, concurrent exceptions occur when some interacting processes encounter each an exception. These exceptions are concurrent as they must be handled, but it is difficult to determine the order of handling and how to synchronize the processes that where initially coordinated along their interactions. Autonomy adds uncertainty in the interactions: Agents can refuse to participate in the handling of exceptions encountered by others. In other words, exception handling mechanisms in distributed systems must be robust to the possible refusal to participate from some agents.

MAS are software systems, so the aforementioned model of exception remains useful to deal with programming exceptions. Distribution and autonomy call for new mechanisms to deal with the challenge they introduce. In particular, the scope of exception is not only the agent process, but also the system as a whole. For this reason, the term of exception management refers to the set of techniques involved in the performance of exception handling in MAS.

1.1 Concepts in Multi-agent systems

This section aims at exposing in detail the concepts introduced in Multi-agent systems (MAS) and the assumptions that define the present work. It develops the notions of agent, interaction, protocol, autonomy, openness, and heterogeneity, and it presents consequences on dealing with exceptions in MAS. The reader already familiar with these notions may skip the whole section as it merely presents 'fundamental knowledge' about MAS. The main information to retain from this section is that *agents are supposed autonomous and interacting according to protocols, in an open and heterogeneous environment.*

Agent. Almost two decades of research in the field of Multi-agent systems and half a century of Artificial Intelligence allow to sketch the notion of agent and to give a consistent definition throughout this document.

Definition: *Agent*

An *agent* is a processing unit in Multi-Agent Systems. It is autonomous and situated in an environment it can locally interact with.

An agent is first a processing unit that executes in the system to accomplish some activities. An agent is in general a *process* that is able to change dynamically its state and the way it changes its state [71, 70]. The manipulation of the state is usually named as a *behavior* of the agent. This characteristic is important to distinguish the original concept of object[1] in Object-oriented programming from the concept of agent as a processing unit. An object has a dynamic state, but its behavior is statically determined at design time, when its *type* is defined (see, e.g., [28, page 16]). In addition, an agent is a software process and it can rightly be implemented as a multi-threaded application, either by mean of object technologies or by other programming paradigms. Agents range in concrete applications from artificial ants [19, 71, 8] to complete software systems [113, 109], provided they satisfy the conditions of interactivity and autonomy. Two types of agents are often distinguished, namely rational and reactive agents. Rational agents contain explicit and symbolic knowledge and a 'general-purpose' reasoning mechanism (e.g. inference engine) to process input with this knowledge. On the other hand, reactive agents have implicit knowledge and usually predefined and application-dependent

Figure 1.2: Picture for an agent throughout the chapters

[1] The concept of object has evolved with the ideas of 'active object' and message-passing techniques. These evolutions of the base concept tend to blur the difference with agents.

process of input. Rational agents are the type exploited in this document as they are seen as more versatile, however more complex.

Interactivity and protocols. Interactivity emphasizes that agents perceive and act upon resources (database, services) and other agents through the environment[2]. Agents interact in a variety of manners, either direct or indirect. Direct interactions are most notably message-passing, where messages are expressed with an Agent-communication language (ACL) [23]. Indirect interactions are represented by communication with tuple spaces [30] and blackboard architectures [81]. Interactivity is essential to MAS, since it is the 'glue' among the agents to cooperate, compete, or converse, to name a few techniques involved in the agent activities [73, 74].

Interactions are usually organized in *protocols* that define the circumstances where agents should interact. The circumstances for an interaction are a purpose, a set of roles, and a sequence of actions (e.g. message exchanges). The *purpose* is the rationale of the protocol, i.e. the expected outcomes of its performance. *Roles* are the functions of agents in the protocol, and *sequence of actions* are the actions that each role can take in the frame of the protocol. Each agent plays a role in order to fulfill the purpose of the protocol. The circumstances of each agent define then the actions an agent can take to comply with the protocol.

For example, checking out a shopping cart in a department store follows an interaction protocol between two agents, each with either the role of client or checkout operator. The operator sends a greeting message at the beginning of the protocol to invite the client. The client sends an acknowledgment message and gives the contents of the cart to the operator. This daily-life protocol continues until the client pays the operator and says goodbye. Artificial agents often interact in the same way as such protocol. The main difference is that human have the capability to be very flexible and to dynamically adapt a protocol slightly, so that to bypass some difficulties or unexpected events, and still comply with the constraints of the protocol (message order, timeouts). It is in general very difficult to have such flexibility in the behavior of agents. Interaction protocols are therefore one of the motivation for exception management. The challenge is to allow agents to cope with some unexpected situations encountered in the execution of a protocol, yet complying with its usually rigid constraints.

Agent technologies proposed alternatives to protocols so as to make agents interact. Planning and computational argumentation models are notable examples. Planning is the dynamic creation of a plan to achieve a goal or perform an activity. A plan is a sequence of actions to perform or sub-activities to complete. For the TeamCore research group, a plan is a well-defined series of activities (go to a meeting point by date t, wait for team members at position x, y) that team members have to perform [49, 50]. A protocol is similarly a sequence of message sending

[2]In much research, the environment is considered as 'transparent', i.e. an identity function. In this document, the more general stance of having an explicit environment is taken [78, 79, 77]. This assumption has no consequence on the work presented here.

actions in the usual case of interactions in MAS. Plans and protocols are therefore close notions, and the present document focuses on protocols. Computational argumentation is much less structured than protocols, owing to the way humans argue in practice [32]. Argumentation models are usually more abstract and flexible, in the sense that the constraints over the agents are weaker than for protocols. They are also more difficult to engineer due to this flexibility. In this state of the research, the present study of exceptions focuses on protocols. Argumentation models could be considered in the future as ways to reason on protocols and possible exceptions.

Autonomy. The second characteristic of agents is autonomy. This notion is elusive as it is difficult to define in a disciplined and concrete way. Many interpretations were proposed depending on the context of application [37, 14, 13], though often seen as the 'absence of global control' [93]. Practical examples of autonomous agents in the MAS community are auction agents that execute on behalf of their owners, following predefined strategies [12, 103, 111, 52].

Formal definitions in dictionaries state the following for autonomy: 'the quality or state of being self-governing; especially : the right of self-government' and 'self-directing freedom and especially moral independence' (from the on-line Merriam-Webster dictionary). In the case of artificial agents, autonomy is here seen as a more pragmatic concept.

Definition: *Autonomy*

Agent *autonomy* is the capacity to decide independently from other agents, and to own a control flow and private data.

An autonomous agent is then a process that is able to evaluate its input and to produce output independently from other agents. In particular, an agent can decide the circumstances of interactions, i.e. the conditions by which the agent will decide to interact with others. The ownership of own control flow and private data is essential to autonomy: Without this ownership, an agent cannot have the guarantee that control is never taken over by another party, even temporarily. The private data contains the knowledge of the agent and its other state information, so that the absence of this type of data prevents autonomy, since such an agent would have no consistency. Autonomy is therefore related to the encapsulation of agents, similarly to the object encapsulation. The autonomy guarantees however a stronger notion of encapsulation to agents, since they have the capability to choose dynamically whether to grant access to the encapsulated information.

Beyond this base definition of autonomy, MAS research has proposed models to describe the relationships between agents. These relationships are directly related to autonomy, since they typically allow agents evaluating their social and resource dependences toward other agents, and thus to modify their autonomous decisions

accordingly [92]. Social autonomy is the degree of independence of an agent toward others within a social model. Social autonomy is often defined against an organizational model, such as the typical hierarchy found in governments and companies. Although agents are autonomous, the organization weaves power and other social relationships that can impact the agent individual autonomy: Agents usually comply with orders issued by agents higher in a hierarchical organization, even though they would have refused to execute the order without the power influence. Resource autonomy is similarly the degree of independence of an agent toward resources. The agent execution usually requires resources such as databases, but also processor time and memory. Some agents need to acquire a number of resources so as to complete their tasks, and resource dependency typically impacts the agent autonomy: An agent must sometimes accept external proposals in order to acquire a resource and complete its task, whereas it would have acted differently if the resource access was granted in the first place. In both cases of social and resource autonomy, the agents are autonomous, which means they can evaluate independently input, output, and interactions. The difference with the base definition is that the social and resource factors modulate the autonomy as they influence the choices of the agent.

Agent autonomy has another consequence on MAS that matters with regards to exception management. It emphasizes the *decoupling* of agents and the *modularity* of the system—'autonomy [...] becomes an additional dimension of modularity' [114]. Both properties result from the definition of autonomy that ensures the encapsulation of agents. They are of direct importance as they are usually wanted in traditional exception handling and other fault tolerance techniques. They contribute to the robustness of software architectures as the propagation of unwanted events such as errors does not spread to the all system, but to some 'modules', i.e. sub-part of the system. Another reason of the importance of these two properties is their relation to open systems.

Openness. Open systems are commonly defined as 'system[s] allowing hardware and software from different manufacturers to be used together seamlessly' (from the on-line Wiktionary dictionary). In the agent community, the meaning of openness is rather akin to system theory, as can be observed in Physics and Management: Energy, resources, or materials flow in and out the system freely. MAS follow this latter meaning for agents in the system.

Definition: *Openness*

A MAS is said *open* when agents can enter and leave the system dynamically.

Openness is a technical challenge as the software architecture of the system must be robust to the addition and subtraction of some of its parts at runtime. MAS

(and related types such as Service-oriented architectures) provide such kind of robustness inherently in theory, owing to the decoupling and modularity aforementioned. The technical concerns are however to ensure the interoperability of the agents, their coordination, and their life-cycles. Interoperability and coordination is addressed in MAS by the adoption of interaction standards, such as the ones from the FIPA that define Agent communication languages (ACL) and services for agents to coordinate (Directory facilitation to discover and connect to agents in the system) [23, 25]. The life-cycles of agents are also defined by standards that list the possible states of an agent and the transition between these states. The FIPA also defines these states and their evolutions with the Agent management system specification [24]. This specification is important for the system openness, as it details how agents enter or leave the system, especially in the case of agent mobility.

Openness is therefore an important characteristics in the design of an exception management system for MAS. Such system must deal with the entrance and exit of agents. It must notably be flexible regarding the number of agents that are involved in the management process.

Heterogeneity. Heterogeneous systems consist of elements that are built according to different design choices. An heterogeneous system can then be made of pieces in different programming languages or it can be developed by different designer teams. The definition in MAS is then:

Definition: *Heterogeneity*

A MAS is *heterogeneous* when agents or the infrastructure of the system are developed by different means.

Heterogeneity makes agent interoperability more difficult and the standards presented in the previous paragraph were designed to be independent of the underlying architecture or language chosen to implement it and the agents. Such standards allow then heterogeneous agents to interact, provided they comply with the interoperability specifications. Another solution that has been suggested is the use of *middleware* components, similarly to the approaches adopted in Distributed computing such as CORBA [16]. Tuple spaces, tuple centers, and more generally the notion of environment are some instances of such middleware in MAS [60, 20, 107].

As for exception management in MAS, the consequence of heterogeneity is that an exception management system cannot always assume that agents are collaborative or even benevolent. This assumption can be reasonable when the designers belong to the same team and follow common design guidances. It is not reasonable when designers are free to choose how agents react to some input and the only requirement is to comply with an interoperability standard. An exception manage-

ment system must then be robust to unpredictable behaviors from agents, notably the refusal to participate in the management process.

Heterogeneity leads then to one of the most general case of exception management, i.e. the *non-collaborative case*. Although collaboration is often a reasonable hypothesis when developing a heterogeneous software, collaboration occults several problems, including that 'perfectly collaborative agents' can *fail* to do as expected in the collaboration and thus showing unwanted non-collaborative behaviors (e.g. to be late). Non-collaboration is therefore a more promising target for an exception management system in MAS. By assuming that agents are non-collaborative in the first place, an actual collaboration can just help improving the management techniques, such as the convergence speed and accuracy of the technique.

1.2 Purpose & Scope of this document

Multi-agent systems are recent software models and techniques that require further research to develop their resilience and their degree of dependability. One way to this end is *exception management* and this document is devoted to this particular topic in the case of knowledge-based agents.

The purpose of this work is twofold, first to study the notion of exception in MAS so as to identify the research directions that need to be followed. The second purpose is to explore some of these directions, notably the ones compiled in the following list.

Concept of agent exception. Research on exception management in MAS has developed the intuitive idea that the concept of agent exception is akin to, but differs from, the usual model of programming exception presented in Fig. 1.1. This intuition originates in the work existing in Distributed computing and the case of MAS introduces new challenges [29]. *The first research issue is then to define agent exception and to relate it to programming exceptions.*

Execution model. Agents are the processing units of MAS and the aim of this document is to endow them with exception management capabilities. A model of execution is at the root of these capabilities to detect exceptions and prepare an agent for their management. The main concerns are to deal with the autonomy of agents, the openness of the system, and the heterogeneity of the system parts. In addition, Software engineering practices recommend a separation of concerns between the main application logic of a system, and its exception handling logic. The separation should appear in the execution model to let designers build systems based on the model, where the code for exception handling is independent from the functional code of the application. *The second research issue is then to develop an execution model of agent that deals with the characteristics of agents and software practices.*

Architectural considerations. Agents are usually implemented as (finite state) transducers, i.e. they transform an input into an output according to some

internal relation. The execution model describes how agents execute and can deal with exceptional situations. The architecture of the agent is elaborated from this execution model to support software designers in implementing 'exception-ready agents'. *The third issue is then to produce a software architecture of agent that supports the exception management and separation of concerns.*

Agent exception handlers. Handlers are methods for agents to deal with exceptions. Several handlers are necessary for an agent as exceptional situations occur usually in various circumstances that may require different handling. In particular, agents are autonomous and they should be able to cope with exceptions by themselves; but they also are part of a MAS, and they may need to collaborate with other agents to cope with some exceptional situations. *The fourth and last issue is then to create generic handlers for agents and guidelines for domain-specific exception handling.*

The approaches proposed in this document to address these research issues, set forth results and techniques that are expected to serve in the agent-oriented computing community, and, eventually, to serve in general Software Engineering, perhaps under an evolved form.

1.3 Case study

The presentation of this document is organized in relation to a case study that motivates and illustrates in a concrete example the model of exception management in Multi-agent system. This section aims at describing the requirements, early design and analysis phases for this case study. The reminder of this document will go through the subsequents stages and refinements of the development process, down to the implementation in chapter 5, with particular focus for the quality requirements addressed with our exception management approach.

Choice of the case study. The case study is a market of rational agents, where agents act and compete in the market on behalf of human owners. Each agent is supposed developed by a different and independent designer for the human owner. The choice for this case has been made based on the recognized applicability and contribution of MAS technologies to market-like systems [12, 103, 111, 76]. Also, this case has the properties of interest in MAS, while still remaining practical for experiments. The properties of *openness*, *heterogeneity*, and *interoperability* are therefore present in the system. Autonomy of the agent is guaranteed by the model and architecture developed along this document, which were designed so that to cope with the aforementioned properties.

The case study serves the essential aim to *validate* the model and *evaluate* its computational complexity.

Settings of the case study. The case study is a market-like system where three types of agents conduct their business, namely energy providers, machinery assemblers, and machine parts providers.

Energy providers produce energy (e.g. petrol, electricity), sell it in the market, and buy machines and replacement parts necessary to conduct their exploitation.

Machinery assemblers build machines for energy exploitation and sell them in the market. They need buy energy and machine parts to conduct their business.

Machine parts providers build machine parts and sell them in the market. They need buy machines and energy to conduct their production.

The description of the agent types reveals resource dependencies. Although the agents are autonomous, these dependencies will lead them to interact to continue their respective activities. In other words, the dependencies are rational incentives for agents to interact with one another. Fig. 1.3 depicts the dependencies and their contents.

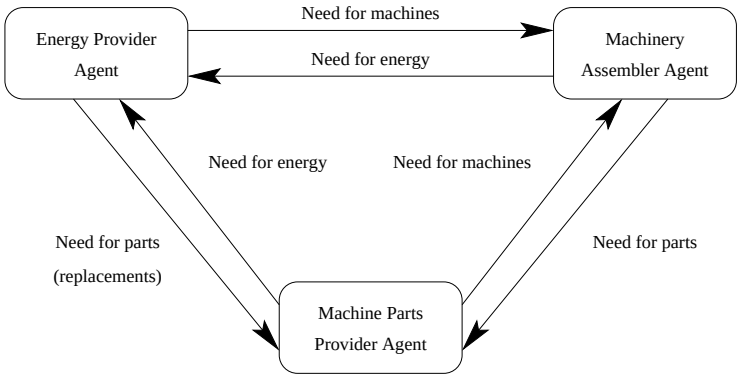


Figure 1.3: Dependency network among actors on the market.

When agents need to trade items, they use the classical Contract Net protocol (CNet) [95] and directory facilities in the system to discover dynamically clients and providers. The CNet is an interaction protocol that has been standardized by the Foundation for Intelligent Physical Agent (FIPA) [27]. Fig. 1.4 represents a version of the CNet adapted to the case study.

The CNet features two roles. The client is the initiator of the protocol and it is the role of the agent that wants to buy items. The provider is the participant role for agents who can sell items to the client. The CNet allows only one client for several providers., i.e. the client calls for proposals from providers to compare prices and choose the best offer. The syntax of the graphical notation is related to UML sequence diagrams [104] and AUML [1], but it has been reduced to the minimum required to describe the case study.

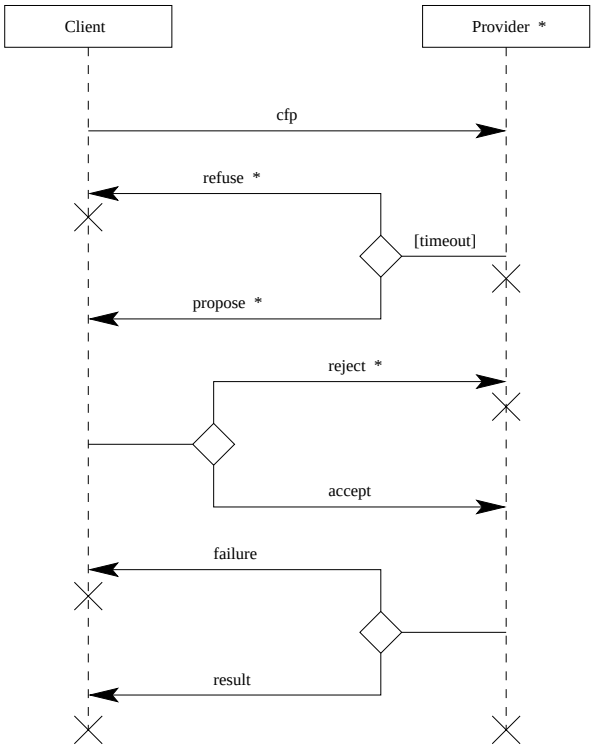


Figure 1.4: Contract net protocol in the market.

Box. A box represents a role and contains the name of the role.

Vertical dashed line. The vertical dashed line represents the execution of a role, where execution is performed along the line flow downward.

Cross. The cross along the vertical dashed line represents a termination of the role.

Arrows. Arrows represent the sending of a message from the agent playing the source role to the agent playing the target role. Interactions are supposed asynchronous as a result of the message-passing model among agents.

Arrow labels. Labels are the type of message that agents can exchange.

Arrow bracketed labels. Bracketed labels are conditions for sending a type of message.

Diamond. Diamonds represent a choice between the sending of several types of message.

Star character. The star character represents items that can have several instances in the protocol.

On Fig. 1.4, the client sends first a call-for-proposal (*cfp*) to selected providers. Providers have to answer before a deadline (t*imeout*), otherwise their participation in the protocol is over (indicated by the cross). Each provider can send either a *refuse* or *propose* message in reply to the *cfp*. The refusal causes the end of the protocol for the corresponding provider. If all providers refuse the *cfp*, the client also stops its participation in the protocol, which is then terminated. Proposals allow the client to continue the interaction to *reject* proposals and *accept* only one of them[3]. Rejection of the proposal of a provider leads it to stop its participation in the protocol. Acceptance elects the provider that wins the *cfp*. At this point, the contract between the client and the provider is settled and the client pays the provider. Finally, the provider fulfills the contract. It can send to the client a *failure* or a *result* message depending on the case. After reception of one of these messages, both remaining roles terminate.

Functional requirements. The simulation aims at having agents conduct their business as long as possible on behalf of their owners. They are initially given a capital to produce their services (among energy, machines, and machine parts) and to buy what they need to continue their productions, as defined in the settings of the case study. An agent is considered 'out-of-business' as soon as it cannot continue its business, i.e. when the agent has no more capital and no service to provide. The value of services is constrained by an 'offer & demand' law that leads agents to increase prices when demand is high, and to reduce them when demand is low. The rationale for this law is to reproduce non-linear dynamics in the system.

Agents are allowed to trade with one another according to the CNet protocol. Openness is due to agents that leave the market by lack of capital (the case of entrance of agents is not considered in this scenario). Heterogeneity is due to agents having to comply solely with interoperability matters.

Quality requirements. Quality requirements are relative to each agent. Each agent represents an autonomous stakeholder in the market, as representative of the human owner. The requirement for each agent is to maintain its activity in the market, i.e. to increase the capital and at least to avoid bankrupt. In other words, each agent is expected to be reliable and available to trade in the market. To this end, agents should not fail in case of problem in the negotiation with other agents or in their productions. One way to achieve this quality requirement is to deal with exceptions in the protocol.

A number of agent exceptions can occur in these settings. For example, a `DelayAnnouncement` exception announcing a delay in the reply or an offer is likely to occur in the market. An agent might wait for the result of one protocol to determine the commitment in another. It can then ask for delaying its answer to

[3]The client could accept several proposals and deal concurrently with each of them. The protocol limits to only one proposal for simplifying the application, as this simplification does not reduce the value of the example as for exception management.

the second. When an agent receives a delay announcement, it can react in different ways, i.e. handling such exception is mostly domain dependent. In the context of the CNet, possible ways to handle it are for instance:

- If a provider of the CNet receives the announcement, it can accept or deny the delay.
- If the client of the CNet receives the announcement, it can:
  - Announce to some or all providers a time extension.
  - Ignore the delay and continue the CNet with other providers.

The chosen way to handle the case is clearly dependent on the autonomous decision of agents and their situations in the environment (e.g. delays are not acceptable with some raw materials).

Another type of exception that should be managed by agents is the case of the 'agent death' [53], which occurs when an agent prematurely terminates and cannot participate anymore in running protocols. Similarly to delays, handling the agent death has different ways.

- If a provider of the CNet is informed about the death of the client, it should stop its participation in the protocol.
- If a provider of the CNet is informed about the death of another provider, it can simply ignore the event.
- If the client of the CNet is informed about the death of a provider *before* contracting, it can simply ignore the event.
- If the client of the CNet is informed about the death of a provider *after* contracting, it has to cope with the loss of money and the need for another contract.

The case study serves along the document to illustrate the model and architecture. In chapter 5, experiments are eventually conducted to validate and evaluate the overall approach, by comparing market runs with activated and deactivated exception management system.

1.4 Organization

The organization of this document has been designed to be progressive regarding the research issues. First, chapter 2 presents research and major techniques akin to exception management in software. This overview of the existing mechanisms aims at describing the current achievements in the various existing approaches and context. The presentation of the mechanisms also exposes the strength and weakness relative to the requirements for agent exception management.

Chapter 3 is an extensive study of the meaning of exception in Multi-agent systems. The chapter aims at defining the expression 'agent exception' and explaining the relationships and differences with the concept of programming exceptions. The study results in an 'exception space' that classifies exceptions depending on their impact (code or agent), scope (one or several agents), and sources (known or unknown). The exception space serves subsequently to define handlers depending on how they address the resolution of an exception.

Chapter 4 develops an execution model of agent exception and a software architecture to implement it. The execution model describes in detail how agents execute and how they deal with exceptional situations. The fundamental idea of this model is for the agent to generate expectations that, if not fulfilled, allow to detect exceptions and engage their handling. The architecture describes a pattern to implement the execution model in an acceptable way, depending on the application requirements. The main contribution of the architecture is to separate at the architecture-level (that is early in the software development process) the mechanisms for the application logics from the mechanisms for the exception logics.

Chapter 5 presents an evaluation of the model through experiments conducted on the case study. The implementation of the system is presented with further details on the experiments settings such as the number of agents and the items they trade. Runs of the system first aim at validating the model and show how agents can deal with exceptions. Runs also aim at comparing the computational cost of the execution model (and the architecture) to the same system running without exception management or with different approaches. The chapter continues with an analysis of the experimental results and a discussion of the approach.

Finally, chapter 6 concludes the document by setting forth the contribution of this work in different target disciplines. MAS are in fact related to Artificial Intelligence and Software Engineering. Exception management relies on and contributes to these two domains of Computer Science and the present work is integrated in their perspectives to emphasize the contribution of the work. The chapter finishes with the presentation of future work that can be derived from the current achievements.

Two

Exception Management in the Literature

Exception management is a research theme that pertains to practical and theoretical techniques in Software engineering (SE) and Artificial intelligence (AI). The purpose of this chapter is to review the achievements in these two domains, and to emphasize their shortcomings in dealing with exceptions in Multi-agent systems. This study is not intended to be exhaustive on the subject, as it rather aims at introducing the most representative techniques that can contribute to exception management in MAS.

Existing work in SE and AI deals with exceptions under many perspectives on software systems. Programming language research was among the first to address explicitly the concern of exception management, in the aim to build more reliable software, more easily for the programmer. Since the early work in the 1970s, the question of exception has followed the evolution of software with new challenges to solve. Exceptions has been under active studies ever since in Distributed computing, Software architecture and Component-based development, Formal models of computation, notably in Logics, and finally in MAS research.

The chapter adopts a presentation method to expose the research consistently in all domains, except the first review of programming exceptions (section 2.1), which presents the work as a whole due to the historical background and aim to present the general exception handling techniques. For all other sections, the scope and assumptions of the research are first presented, followed by a detailed description of the approach, and a discussion of the contribution to exception management in MAS. The chapter concludes with an overview of all the achievements, a summary of the research directions that are left open by these achievements, and an introduction to the directions that are addressed in this document.

2.1 Exceptions in programming languages

The original motivations for exception handling in programming languages are due to the context of the 1970s. Hardware then suffered problems of reliability and the development of reliable software was lacking systematic techniques, as can be observed in the history of exception handling mechanisms [35, 33, 34]. Original techniques to deal with exceptional situations were either ad hoc or complex to exploit, at least from the perspective of the present achievements in programming languages. The purpose of an exception handling system was then to have systematic treatment of exceptional conditions to either recover an appropriate program state and resume the execution, or to terminate the software 'gracefully', i.e. to ensure there is no side-effect in stopping the execution (e.g. release only reserved memory, persistent data consistency).

From the original work of John Goodenough in the 1970s, most implementations of exception facilities in a programming language follow similar semantics, with slight differences depending on the constraints of each language paradigm (typically procedural, functional, object). From languages as CLU to PL/1 to ML to Java, the way to handle programming exceptions principally differ in the language syntax. Some other languages have introduced different additional mechanisms[1], such as LISP.

This section of the related work presents first the common semantics of exception handling in programming languages, i.e. the sequence that is executed by the software at runtime when it encounters exceptional conditions. This section also presents the case of LISP to show that there exists different ways to deal with exceptions, despite the overwhelming success of the main-stream approach, due to its simplicity and the qualities of implementations that can be found. The section concludes with a discussion of the relation to MAS.

2.1.1 The original semantics of exception handling

Most languages rely on a similar model of exception that was introduced in Fig. 1.1 (page 4). When a program is in execution, the invocation of an *operation* can encounter an *exceptional condition*. The execution flow is then deviated to a *handler* that deals with the condition, until it resumes or terminates the execution of the program. An operation is any instruction or set of instructions that is called for execution. Before performing the actual operation, a set of pre-conditions is checked to ensure no harmful execution can occur (e.g. committing to divide by zero could have disastrous side-effects on the computer memory by erasing or overwriting some areas). If one of the pre-conditions is not verified, the program is

[1] The supplementary mechanisms of these languages were often suggested by Goodenough in its original papers. The common exception handling system actually implements for the major part the essential subset of recommendations from Goodenough [34]. The recommendations that are usually escaped are the ones dealing with *monitoring* that serve to take some 'fortuitous' initiatives in the execution (react to events that are not only failures) and are therefore complex in use.

said to encounter an 'exceptional condition', since the invocation of the operation assumes that all conditions should pass. A handler is then searched: It is a block of code that contains a series of instructions to deal with the exceptional condition.

The search is performed according to the program and the current execution. Handlers are associated to a syntactic unit in the code, which is an instruction or a block of instructions. Exceptions occur in a syntactic unit and handlers are first searched in this one. If no handler is available where the exception has occurred, the handler search continues by requiring the handlers attached to the syntactic unit of the previously executed instruction, which is found according to the call-stack maintained by the program. This search is called 'unwinding the call-stack'. The following code on Fig. 2.1 illustrates the syntactic units and the unwinding.

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;

class SemanticsException {
 public static void main(String[] args) {
  System.out.println("Start example...");
  final SemanticsException se = new SemanticsException();
  try {
   se.process();
  } catch(IOException theIOException) {
   System.err.println("***File cannot be read!***");
  }
  System.out.println("Finish example.");
 }

 public void process() throws IOException {
  try {
   final FileReader lfReader = new FileReader("file.dat");
   final char lfChar = (char) lfReader.read();
   System.out.println("First character: " + lfChar);
   lfReader.close();
  } catch(FileNotFoundException theFNFException) {
   System.err.println("***File does not exists!***");
  }
 }
}
```

Figure 2.1: Sample Java code to illustrate syntactic units and handler search.

The `try catch` keywords in Java allow to define syntactic units where handlers are attached. In the above example, the `main` method features a handler for `IOException`, and the `process` method has a handler for `FileNotFoundException` and can propagate (`throws`) `IOException` to the calling method. In the code, the instructions `read()` and `close()` can fail

and signal the `IOException`, while `new FileReader` can fail and signal `FileNotFoundException`. When the piece of code is executed while the file named 'file.dat' does not exists, the instruction `new FileReader` fails and searches for a `FileNotFoundException` handler. As the syntactic unit is then the `try` block in the `process` method, the code shows that a handler is available and the exception can be handled locally (relative to the execution flow). On the other hand, we can assume that the file does exist when `new FileReader` is called, but it is erased before the call to either `read()` or `close()`. The program then searches for an `IOException` handler along the call-stack. The `process` method explicitly propagates the handling of this exception to the caller, which means the handler must be provided in the syntactic unit of the caller (which can also propagates it). The caller is the `main` method, which provides the handler in the syntactic unit defined by its `try/catch` block. The exception is therefore handled there.

The general search mechanism terminates when an appropriate handler is found, or when the call-stack is totally unwound, which means the search has failed and the program must be abnormally terminated ('unhandled exception' error). In case a handler is found, the handler can either *resume* or *terminate* the execution of the instruction (or block of instruction), depending on the operation and the language implementation. Once the handling procedure completed, the execution of the program can continue, but the data in the block that has been interrupted is lost. The state-chart 2.2 summarizes the description of the common operational semantics.

The state-chart shows on the left-hand side (white part of the chart) the expected procedure to execute an operation when all its conditions are met. The right-hand side shows the typical procedure to cope with exceptions (gray part of the chart).

Most languages follow a semantics close to this one, notably Java [36], C#, the different versions of C++ [99], ML, and so forth. The respective homepages of these languages give all the necessary details to understand the actual semantics in these language implementations. Slight differences can occur depending on the language paradigm or design choices. For example, the Visual C++ from Microsoft features a 'structured exception handling' mechanism that allows a closer collaboration between the program and the operating system to deal with exception conditions [69].

2.1.2 Alternative model: Condition handling in LISP

Exception handling in LISP is more general than the previous model and it is also closer to the original proposal from Goodenough. The handling system is there called 'condition handling system', where a condition is a generalization to any event, either error, *exceptional* situation, *or else* [89, Chapter 19]. That is, conditions stand at a higher level than exceptions, so that any event can be handled the same way.

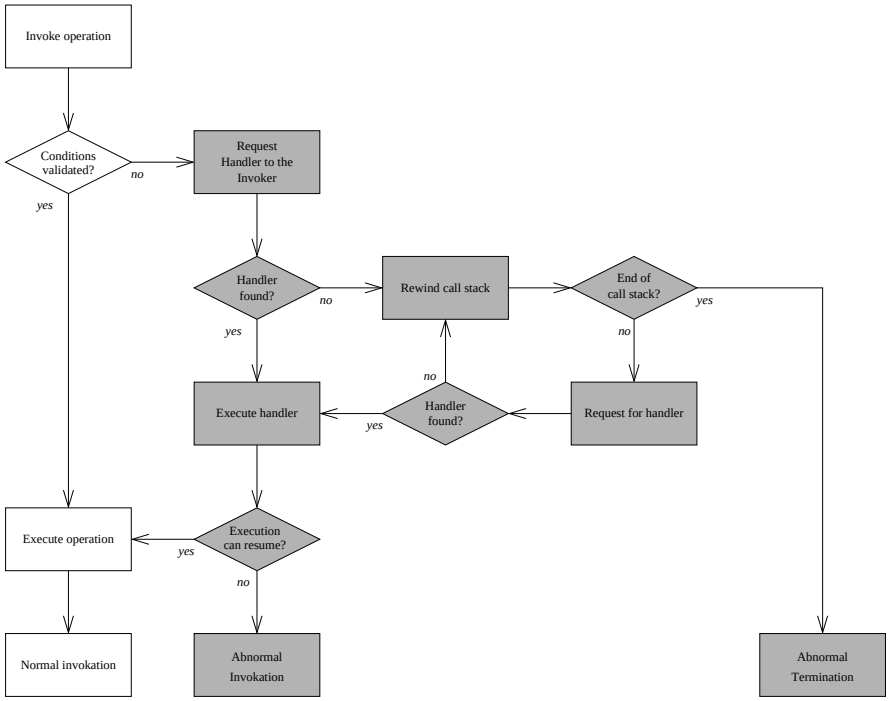


Figure 2.2: State-chart describing the operational semantics of exception handling in many programming languages

The main difference in the semantics holds in the distinction between signaling, handling, and restarting in the process of dealing with a condition. The previous semantics focuses on the two first mechanisms only. In addition to the procedure shown in Fig. 2.2, the condition system of LISP allows to define 'restarts' instead of handlers in the program. When a condition is encountered in the program and a restart is triggered, the call stack is not unwound, which means the program does not terminate and it has the possibility to continue its execution without loss of data. The restart defines a handling procedure for the condition and the point were the execution should resume once the procedure has completed.

2.1.3 Programming exceptions and agents

Programming exceptions pertain to conditions in the execution flow of a program. The level where exceptions occur is therefore instruction-wise, which has an accurate meaning in the code. Agents are first of all software programs, so this type of exception does matter and should be handled as in any program with the current state-of-the-art handling systems.

However, agents are autonomous software that process events in their environment. When such event occurs, the agent can encounter a situation that is 'exceptional' to the activity executed by the agent, while the event does not cause

any programming exception (c.f. the `DelayAnnouncement` in the case study). The reaction to an exceptional event at the agent level differs from an exception at the code level: There is no call stack to rewind with an exceptional event, which must be handled in the continuity of the agent activity. Also, the call stack rewind is a lossy process, since the context of each call is lost at each step of the rewind. If the agent is to remain *autonomous* in face of exceptional events, handling should occur without loss of data for the agent. The operational semantics of exception handling, and therefore exception management, requires a new model that is adapted to the functioning of agents, and that completes the necessary handling system for programming exceptions.

The observation of a type of exception in MAS that differs from programming exception is originally due to the distributed (or decentralized) nature of MAS. The next section then presents the achievements in the domain of Distributed systems.

2.2 Exceptions in Distributed systems

2.2.1 The Guardian

The 'guardian' is an architecture and a programming model to handle exceptions in a distributed-object system, with applications to the mobile agent context [102, 66]. The guardian is a specific object designed to orchestrate concurrent exception handling. It provides a set of methods to have application objects enter or leave a context under its management, to signal *global exceptions*, and to propagate them. Global exceptions are programming exceptions handled by the guardian. When such an exception occurs, the guardian applies a corresponding *rule* that is defined by the application developer to describe the global handling procedure. The guardian follows the rule that usually entails the enabling of local exception handling in the objects impacted by the global exception. The guardian allows then to recover from exceptions in a distributed way, despite the possible concurrency of exception signals or issues in coordination. Global exceptions are introduced as a complementary model to the 'local exceptions' presented in the previous section. Global exceptions are in fact particular to distributed systems and are not required in sequential systems.

The semantics of the guardian handling process differs from the usual semantics of programming exceptions, due to these global exceptions. Fig. 2.3 shows a state-chart that illustrates this semantics, in comparison to the usual one in Fig. 2.2.

The original exception handling state-chart in Fig. 2.2 is reduced to the 'Usual exception handling' box (dark gray), and the new elements of the semantics (light gray) are introduced between the condition validation test and the usual handling state. The notable difference in this semantics is that the guardian model allows the *continuation* of the execution whenever a global exception is signaled. The guardian model applies 'recipes' to deal with exceptions, instead of a rewind of a call-stack-like structure. This continuation allows to abstract some issues due to

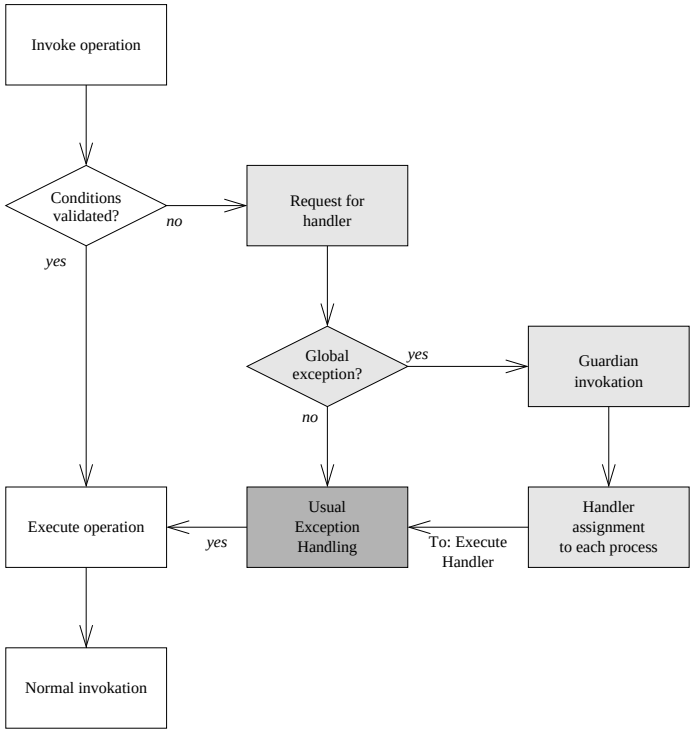


Figure 2.3: State-chart describing the operational semantics of exception handling with the 'guardian'

concurrent exceptions: It would be very difficult to rewind in a coherent manner the call-stacks of a multi-process distributed application. The guardian model bypasses this difficulty by deciding the handling procedure on behalf of all processes involved in an exception. Each process receives from the guardian a 'usual' handler to execute, which is coherent with the handlers that other processes will execute in the overall recovery procedure.

Example. A detailed example of exception handling is presented by Miller and Tripathi where the direct relationship with Java facilities can be observed [66]. The guardian assists a client-server system shown in Fig. 2.4 that implements the 'primary-backup' approach to deal with server-side failures [101].

Clients connect to a server to get some services executed in a usual request/reply fashion. Behind the scene on the server-side, the primary-backup is a replication of the server on another one. The actual server that connects to client is named the primary, and the second is the backup. When the primary executes a service, it modifies its state, delivers the service, and sends the modification to the backup, so that both servers end in the same state after each service provision. Whenever the primary fails and has to terminate, the backup transparently takes over the connection for a seamless continuation of the server activity. Depending

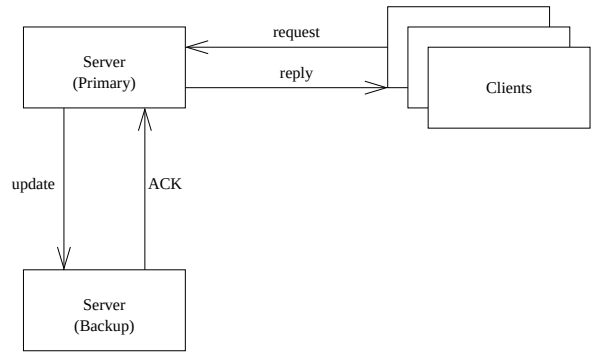


Figure 2.4: Architecture of the 'primary-backup' fault tolerance

on the quality of the swapping between the two servers, clients might not be aware of the failure.

The guardian programming model provides the necessary facilities to implement this approach. The introduction of the guardian yields a new architecture as shown in Fig. 2.5.

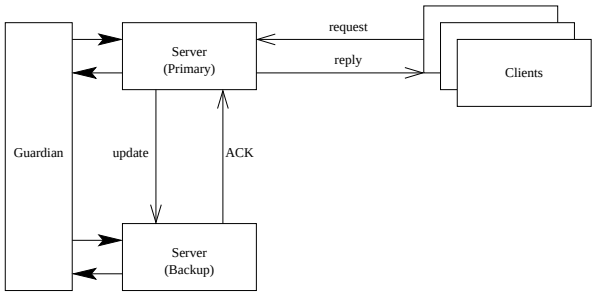


Figure 2.5: Architecture of the 'primary-backup' with the guardian

If the primary server fails, a 'global exception' is raised, so that the guardian handles the error by creating a handler for the backup, which is expecting to synchronize with the primary. The handler requires the backup to take the role of primary, and to instantiate a new backup.

Guardian and agents. The notion of global exception and the corresponding handling semantics confirm the intuition that agents can encounter other exceptions than traditional programming ones. The guardian programming model however assumes that the processes under management are collaborative, and this is a limit of the application in the case of open and heterogeneous systems such as MAS.

In addition, the guardian model ends its handling process by producing and assigning traditional exception handlers to the processes. That is the usual semantics is joined at the end of the global exception handling. In other words, the

processes still lose information in the handling procedure, which is proper to the programming exception models.

2.2.2 Coordinated and cooperation exception handling in distributed objects

Coordinated exception handling and Cooperation exception handling are two approaches designed to deal with exceptions in distributed object systems. The difference with the guardian is that they do not recommend a priori any particular architecture and they elaborate on the 'orchestration' aspect of processes to manage exceptions.

Coordinated exception handling. Coordinated exception handling relies on the concept of coordinated atomic actions and exception graphs to deal with concurrent issues that can occur in the system [112]. Fig. 2.6 helps to illustrate this concept.

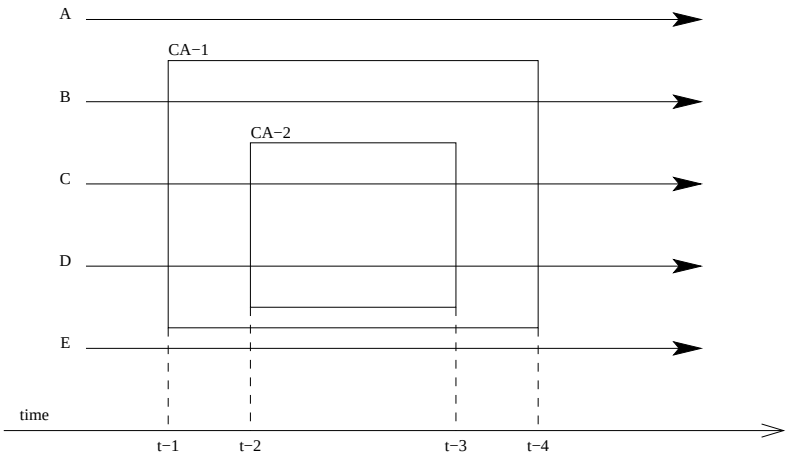


Figure 2.6: The coordinated atomic action model

In a distributed application, one of the main difficulties is to determine which processes are involved or impacted by an exception. Coordinated atomic actions (CA) create a virtual context for a number of processes to circumscribe the group of processes that must participate in an exception handling procedure. When processes belong to a CA, they interact among one another, but they do not interact with some processes out of the CA as long as the CA exists. in Fig. 2.6 processes C and D belong to CA-2, so that they do not interact with other processes in the interval (t-2, t-3). In the intervals (t-1, t-2) and (t-3, t-4), processes B, C, and D interact in CA-1, while they do not interact with other processes. A and E can interact at any time according to this graph, and they can interact with B, C, and D either before t-1 or after t-4. The CA model allows to confine an exception handling procedure to a subset of processes. If an exception occurs in C in (t-2,

t-3), the only process that will be involved in the handling procedure are C and D. If the exception is handled inside the CA, the execution then continues. If there is no handler available, the exception is propagated to the immediately enclosing CA. In the case of an exception in C that is not handled in CA-2, the next attempt to handle it will be in CA-1 with processes B, C, and D.

Another difficulty is to determine an appropriate handling procedure that is consistent with all processes in a CA, especially in case of concurrent exception signaling. Two or more processes can signal different exceptions. A consistent handling must ensure that the handling procedure allows to deal with all the exception types. In the coordinated exception approach, processes can access an *exception graph* that allows to determine a common handler when concurrent exceptions are signaled. Such graph is shown in Fig. 2.7

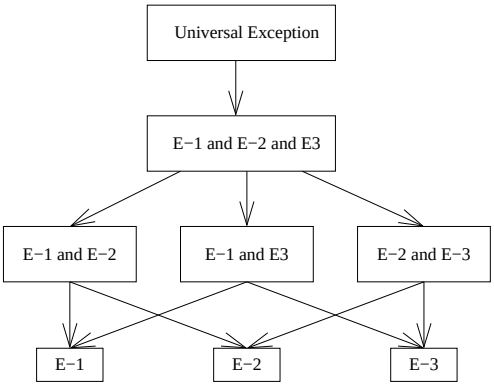


Figure 2.7: The exception graph model (reproduced from [112])

When concurrent exceptions are signaled, they are mapped to the leaves of the exception graph. The common handler that will be selected is the first common parent of the signaled exceptions. For example the signaling of E-1 and E-3 will select the handler of the 'E-1 and E-3' parent in the tree. As with usual exceptions, a universal exception is defined and it matches 'any kind of exception', so that exception types that do not require specific handling or that are not supported by the application in the first place can still get the basic support from the handling system for 'graceful termination'. The coordinated exception handling approach was validated among others on a production cell application, which is the theme of several MAS implementations and can make it relevant in practice [26, 21].

Cooperation exception handling. Cooperation exception handling elaborates on a cooperation model among distributed objects involved in *multi-party interactions* [43]. This work introduced the models of global exceptions and concerted exceptions, later reused in the guardian. The difference with the guardian is that the cooperation exception handling system relies on particular constructs that can be integrated in languages. The model has been implemented in the program-

ming language Arche, which is designed to distribute computing over local area networks. The detailed presentation of this language is out of the scope of this presentation of the work, and it would require a significant space. The exception handling part can however be explained with some simplifications.

The language models a distributed application as *multi-procedures* (MP), which are procedures executed in parallel with a specific semantics for their initialization and termination to ensure proper executions. Procedures declare exceptions they can signal to their enclosing MP for handling. In addition, procedures can explicitly bind their execution to other procedures in order to synchronize with them. Two procedures that are bound then execute in synchrony, and the occurrence of exception is resolved in common. Without entering the details of the language, Fig. 2.8 shows how the binding is expressed in the model.

```
resol rSumFac(handles ov() ; signals ov()) =
  begin throw ov end
mproc fact(v x : int; r fx : int)
  [ov()] = not detailed : fx = x!
mproc sum(v x : int, y : int ; r s : int)
  [ov()] = not detailed : s = x + y
mproc sumFac(v a : int, b : int ; r sf : int)
  [ov()] using rSumFac =
  (v a : int ; r sf : int) [ov()] =
   var f1 : int;
   begin
   {fact(a,f1)}[ov() :  throw ov()];
   {sum(x = f1, sf = s) with 2}[ov() : throw ov()]
   end
  parallel
  (v b : int) [ov()] =
   var f2 : int;
   begin
   {fact(b,f2)}[ov() : throw ov()];
   {sum(y = f2) with 1}[ov() : throw ov()]
   end
```

Figure 2.8: Syntactic bind at compile-time in a multi-procedure (reproduced from [43])

The MP of interest is the last block introduced by 'mproc' named 'sumFac', to compute the sum of two factorials computed in parallel. The MP contains two procedures that compute individually a factorial. When these two procedures add their respective results, they invoke the MP 'sum' with a subset of the required arguments and the special keyword 'with', which introduces the identification number of the process to synchronize with. The two processes have then safe access to the shared variable of the sum and they are synchronized in case of exception.

The approach appears similar to the aforementioned coordinate atomic action

model. The differences are however that bindings among processes are statically declared in MP, whereas CA can evolve over time, as shown in Fig. 2.6. In addition, the exception graph of CAs seems more generic approach than the introduction of language constructs (the 'resol' keyword in Fig. 2.8). This last comment is however weakened by the lack of knowledge about the implementation of the exception graph.

Coordination, cooperation models, and agents. The coordination and cooperation exception handling approaches explicitly deal with programming exceptions, and the mechanisms based on distributed algorithms and programming seem applicable to MAS, especially the work on multi-party interactions. These approaches cannot be exploited directly however, owing to the assumptions that agents would be cooperative and inspected. In addition, some agent exceptions such as the agent death are not taken into account [53].

2.3 Architecture-level and Component-level exceptions

Research in Software Engineering recognizes exceptions not only in programming language, but also at the level of the system. Exceptions then pertain to a significant part of the system that must react in coordination with other parts, instead of just having a local handling. At the system level, two main areas of work have been developed, in terms of software architecture and component integration.

2.3.1 Architecture-level exception handling

Research in Software architecture proposes exception handling related to architecture description languages (ADL), which target Software Engineering directly at the architecture level. One notable instance is the work of Issarny and Banâtre that introduces exception handling constructs and runtime support to an ADL [44]. The use of this extended ADL allows to specify how the architecture reacts to some exceptions.

Examples of such architecture-level exceptions are related to the client-server architecture. The language allows to specify that a base architecture (e.g. RPC communication) can evolve for dynamic binding of component instances, enhanced availability (replication), or enhanced response-time (pre-fetching), whenever such evolution is necessary to maintain the system performance.

Such work at the architecture-level is relevant to MAS, which are open architectures hosting autonomous agents. However, the extended ADL proposed in current work mostly aims at cooperative components, so that further extensions are required to deal with autonomous entities.

2.3.2 Exceptions in Component-based Software Development

In relation to Software architecture, the development of software based on COTS (Components-On-The-Shelf) aims at building systems by assembling generic 'ready-to-use' components [100]. The issue with COTS in practice is the actual integration of arbitrary components into a robust application. The implementation details of components are usually not known, and only some details about the provided functionalities are delivered with a given component. Integration of components is therefore difficult as 'systemic exceptions' can occur due to their assembling [18]. In addition to traditional exceptions handled inside components as individual sub-parts of the system, system-level exceptions need specific mechanisms, in the same way agent exceptions call for novel approaches.

Sentinel components. Dellarocas proposes a model developed in relation to the work of Klein et al. in MAS [18, 51, 53]. The approach is to introduce pluggable 'sentinel components' in the assembling of COTS and request the components of the application to implement a set of interfaces that lets sentinels detect and deal with exceptional behaviors. Sentinels actively observe the execution system-wide for symptoms and they exploit a knowledge base of handling recipes to recover a variety of situations.

Coordinated exception handling in components. A later approach relies on the Coordinated exception handling approach, aforementioned in the case of distributed systems [112, 84]. The work is a generalization of the atomic action model to components. The execution of components is organized into actions that define a scope wherein exceptional situations must be managed. Action scopes can be nested so that the usual recursive handling schemes are reproduced: An exception that cannot be handled inside an action scope is propagated to the enclosing action scope. The main advantage of this approach is to provide a dynamic mean to organize the execution of components into actions, and to manage the occurrence of concurrent exceptions inside these actions. In addition, this work proposes guidelines to software integrators for introducing this exception handling mechanism in the development process of COTS assembling.

Components and agents. The component-based approach assumes that application components are observable and commandable (through the required set of interfaces), and this hypothesis is not acceptable with agents. In the case of sentinel components, the approach based on system-wide observation does not hold in MAS where agents only have a local scope and scalability issues arise as the number of agents or the complexity of their interactions increase. Finally, the exploitation of a large knowledge base to provide handling recipes is attractive for many practical cases, but scalability issues also arise when the size of the base and the frequency of search increase. The structuring offered by the action model is also not fully applicable in the case of agent exceptions. One of the assumption of this work is

in fact that 'components have deterministic behavior and do not change their state spontaneously' [84]. In other words, components need an invocation to ever react, similarly to an object in Object-oriented programming. Although agents can be predictable, they usually evolve spontaneously as they execute autonomously.

In the case of agent systems, a similar view to components can be observed as putting agents together in a system can be thought of as assembling components into an application. The major difference is however the notion of module. In component-based development, a straightforward definition of a module is a single component and this provides a context inside which exceptions are handled. The respective work of Dellarocas and Romanovsky show this definition of a module only allows to deal with traditional exceptions and let systemic exceptions unaddressed. The notion of action is another definition for a module in the system to deal with such exceptions, as illustrated in the Coordinated atomic action model. It offers the advantage to be more flexible, therefore dynamically adaptable at runtime. Assembling components together results in a tight coupling of modules, where exceptions occur in each module and in the resulting product as a whole.

Two types of 'module' are observed in MAS, namely the agent and a group agents involved in a common activity. Agents are indeed strongly decoupled entities, and that is the reason why the component-based approach does not fully satisfy the case of agent exceptions. The lesson that can be learned for agents is that putting in the same system apparently interoperable agents does not guarantee their proper functioning, especially as agents enter and exit dynamically.

2.4 Exception in Logics

Logics has been extensively exploited in AI to develop agents with cognitive capabilities. Logics allows to represent how agents can 'reason' to execute the work they are submitted with. The mechanism behind the reasoning capability is an inference engine embedded into the agent that derives logical conclusions from a set of inputs including knowledge and changes in the environment. Interesting work in logic-based agents pertains to the reaction to 'abnormal situations' with non-monotonic reasoning techniques such as in default logic, the situation calculus, or the event calculus [65, 54]. Abnormal situations are indeed akin to the exceptions studied in this document.

2.4.1 Default logic and Circumscription

The usual example of formula with an abnormal situation is about birds. Common sense dictates that birds normally fly, although it is not always true, for example if the bird is injured or a penguin. The following formula states that if X is a bird and it has no abnormal characteristic, then the inference engine can derive that X flies.

$$\forall X, \quad bird(X) \wedge \neg abnormal(X) \supset flies(X) \tag{2.1}$$

The problem with this formula is that an inference engine cannot derive any conclusion from the only knowledge that 'X = Tweety' is a bird. The logical formula requires explicit knowledge relative to the $abnormal$ predicate, either $abnormal(Tweety)$ or $\neg abnormal(Tweety)$ in this case to conclude. Several techniques were proposed to deal with this matter, and they are relevant here as they illustrate how logical mechanisms allow distinguishing normal from exceptional cases.

Default logic. In default logic, specific rules are introduced to inform the inference engine about *default*, i.e. assumed, knowledge. The pattern of the rules is $A : B \supset C$, where A is the hypothesis, B the default, and C the conclusion. It is informally understood as if A holds, then C is true whenever the negation of B is not known. For example, the next formula means that if X is a bird, it normally flies by default, so the engine can conclude that X flies whenever there is no predicate that states that it does not.

$$\forall X, \quad bird(X) : flies(X) \supset flies(X)$$

Circumscription. In the Situation Calculus, McCarthy proposed the circumscription of predicate as a logical mechanism to achieve the same as default logic, but with the advantage to avoid using specific rules. Circumscription avoids to introduce a new syntax and relies only on the usual logical operators of first-order predicate logic[2]. The detail and formal mechanism of circumscription is described in the original work [62, 63], and what matters for the engineering of agent exceptions is the informal semantics of the approach. The problematic formula becomes:

$$\forall X, \quad bird(X) \wedge \neg \text{Circ}[abnormal](X) \supset flies(X)$$

Circ$[abnormal]$ is a predicate that formally and concisely enumerates what is known and deems 'the rest' to be false. Circ$[abnormal]$ can be chosen, for example Circ$[abnormal](X) \equiv (X = Donald \vee X = Daisy)$. It means that $abnormal$ is true for the value $Donald$ and $Daisy$, and false for any other value. The inference engine can then derive that $X = Tweety$ can fly, since the only known exceptions are $Donald$ and $Daisy$.

Default logic, circumscription, and agents. Logical mechanisms such as default logic and circumscription demonstrate formal means to model the reaction of an agent to exceptional situations. The main relevance of these mechanisms appears in many models of exception handling in programming language. An inference engine cannot derive any result from formula 2.1 if the only input is the predicate $bird(X)$: The engine may just block or return it cannot conclude. Similarly, a program may just block or exit in abnormal conditions when it encounters an exceptional situation

[2] Circ is a predicate over predicates, introducing a single second-order term (Circ itself) to achieve circumscription of a first-order theory.

without any handler available nor propagation option (i.e. lack of knowledge to be compared to not knowing $abnormal(X)$). The strength of the above logical models is that they provide a formal mean to avoid blocking or exiting, and still keep the agent or program in a consistent state.

2.4.2 Abductive reasoning

The last contribution of Logics to this survey is actually devoted to Multi-agent systems through the use of Abductive logics to reason about failures or speculate about the possible futures [88, 87].

Abductive logics is usually exploited to generate hypothesis about the activities at hands. A hypothesis allows a logic program to execute even though the knowledge lacks proved grounds. The program can then execute speculatively until the target result is obtained, or the hypothesis is proved incorrect. In the latter case, the execution continues with the knowledge that the hypothesis was wrong.

The work of Satoh on failure and speculation shows examples of applications of Abductive logics to cases akin to exception handling [88, 87]. One example is a MAS organized as a hierarchy of agents. Agents on top of the tree receive tasks that can be decomposed into sub-tasks and distributed to lower-level agents in the hierarchy, as shown in Fig. 2.9.

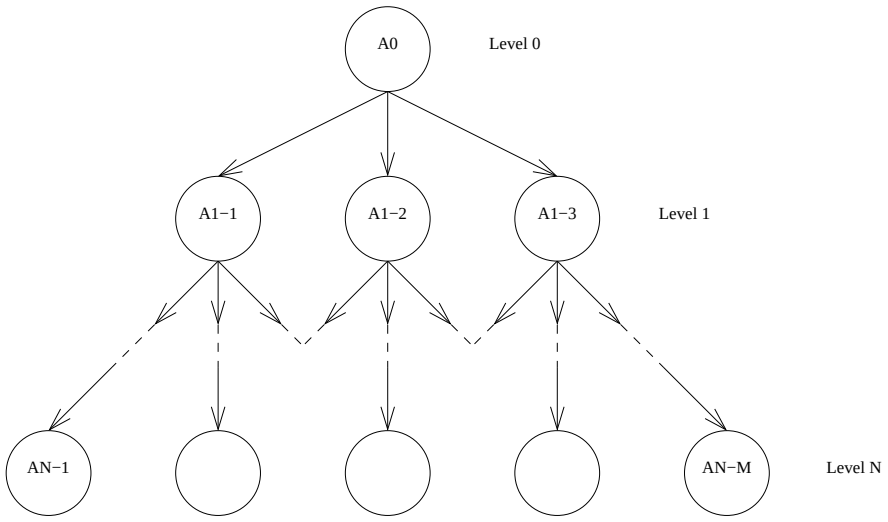


Figure 2.9: Hierarchy organization of a MAS

When an agent at level i receives some tasks, it decomposes them and requires agent at level $i-1$ to perform parts of the decomposition. During the performance of the parts, the agent at level i is *not waiting for the result of each part*. It assumes necessary results as optimistic hypothesis and continues its own execution until the real results are necessary. If an assumed result is eventually received, the hypothesis is replaced by this value. On the contrary, a failure leads the agent at

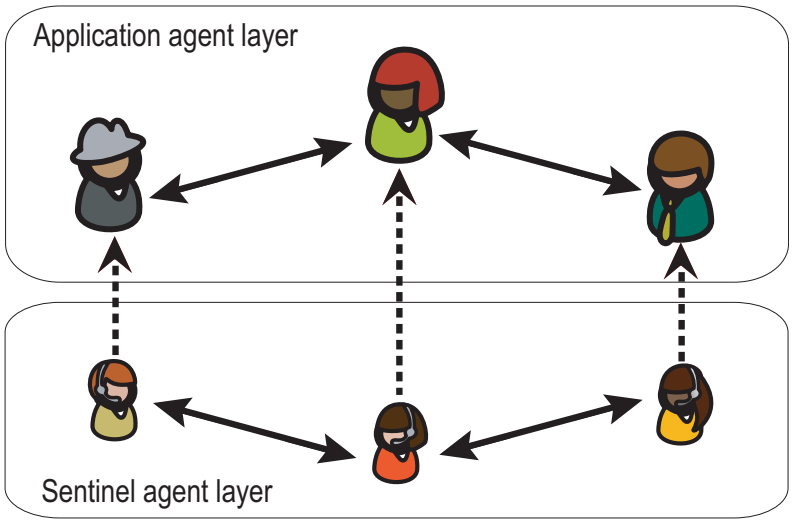


Figure 2.10: Representation of the original sentinel approach

level i to re-allocate the performance of the sub-task, which is a type of exception handling.

Abductive reasoning allows to formally represent this mechanism. It appears particularly useful in the case of MAS where the above situation is likely to occur. In addition, the abductive framework of this particular example illustrates that agents can *individually* reason about their environment and manage a number of exceptional situations *autonomously*.

2.5 Exceptions in MAS research

2.5.1 The sentinel-based architecture

Sentinels are agents introduced in a MAS application to provide the system with a fault-tolerance service layer [40], as depicted in Fig. 2.10. Each sentinel assists an application agent in its interactions with other agents. Sentinels are specialized in error detection and recovery, with the capability to inspect the state of agents (including their 'beliefs' [80]). When an exception is detected in interactions or agent states, the sentinels execute specific code to recover a desired state.

A detailed application from Hägg is the use of MAS in the context of a power distribution company. Application agents negotiate energy consumption credits for load-balancing on the electric grid. Sentinels can detect and remedy to erroneous behaviors in negotiation processes by inspecting 'checkpoints' in the agent code.

The original approach has been extended in the work of Klein et al. with an exception handler repository that provides sentinels with handling recipes inspired by management research [51]. Sentinels can therefore better coordinate to solve or improve the system execution facing exceptions. The advantage over the

original sentinel model is that exception handlers are shared in the repository, so that system designers do not need to produce specific sentinels. The 'handling code' is available to any sentinel whenever required. Another work has extended this approach with a detailed architecture for sentinels devoted to exception diagnosis [91]. This work focuses on analyzing the contents of FIPA-compliant agent communication languages [23]. The analysis is performed by sentinels who also hold knowledge on running agent protocols and plans. Whenever an exceptional situation is detected, the sentinel dialogs with its corresponding application agent to try recovering a consistent state.

The problem with the sentinel approach is that it violates assumptions of the agent paradigm. Encapsulation is not respected since sentinels can access and execute code in the so-called 'agent-head' [40], which should be a black-box to respect agent autonomy. In addition, the latter extension is declared to be part of the hosting system where agents can freely join and leave [91]. As sentinels are allowed to fully inspect agents, this architectural style violates further the assumptions of openness and agent autonomy. Finally, agents are supposed benevolent and this hypothesis cannot hold in heterogeneous systems.

2.5.2 Reliability database and sentinel-like agents

Another version of the sentinel approach has taken a different approach and improves some of the shortcomings, most notably the respect of agent autonomy. Klein et al. proposed to keep the sentinel model of supporting application agents and to complete the system with a *reliability database* [53]. The sentinels function similarly to the original model of Hägg, but they do not inspect agent internals, thus better preserving their autonomy. Sentinels serve as proxies of agents in the system and monitor interactions to provide agents with appropriate interaction protocols when exceptions occur. The novelty is that failing agents are registered in the reliability database to keep track of problems of high frequency. The database guides sentinels in recovery procedures to improve the mean recovery time. The corresponding architecture is presented in Fig. 2.11.

Application agents interact *through* their sentinels to contain any problem and exploit the reliability database consistently.

The approach has however two shortcomings. The agent autonomy is not completely preserved because sentinels are allowed to modify agent messages in two circumstances. Messages can be changed in handling of exceptions to resolve the problem encountered by the agent, and messages can be redirected to more reliable agents according to the database. Although these two changes are acceptable in the context of this research on collaborative agents, it is not acceptable to preserve autonomy. The second shortcoming is identified in the articles of this research: The exception management system is brittle when agents or sentinels fail to fulfill their tasks during an exception handling procedure. This issue is actually one of the base motivations to complete the typically system-level approach with a reliability database with agents that endow individual exception management capabilities.

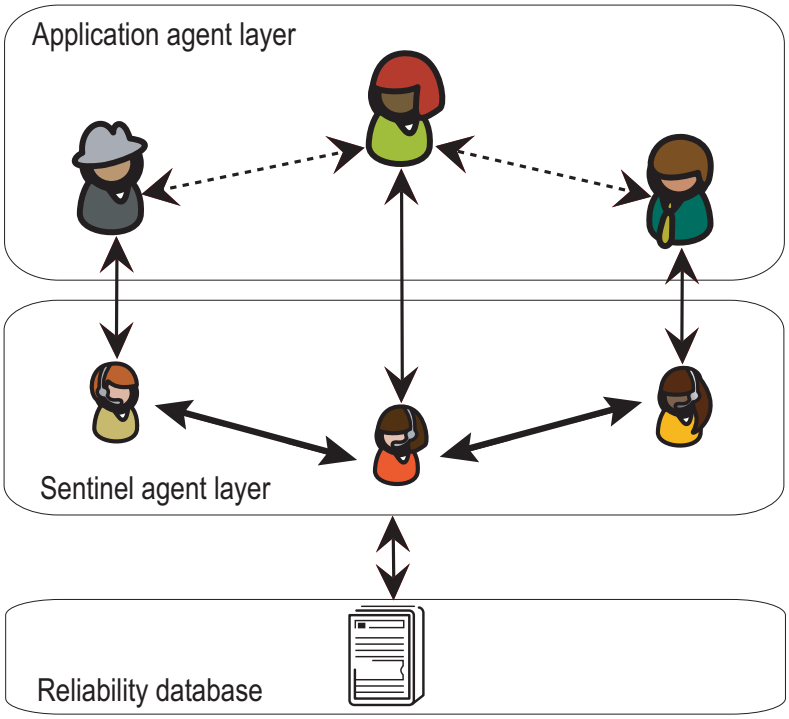


Figure 2.11: Reliability database in the sentinel approach

2.5.3 Agent exceptions in commitment protocols

The work of Mallya and Singh deals with exception handling for autonomous agents in the context of business process [59, 58]. This approach relies on commitment protocols to specify how autonomous agents interact in an open system. Commitment protocols are interaction protocols whose formal semantics aims at better representing the social commitments of agents when they engage in a protocol. As for exception management, the advantage of commitment protocols is to better preserve the autonomy of agents.

When an agent detects an event that does not follow an agreed protocol specification, it considers the event as an exception and two mechanisms formally defined allow to handle expected and some unexpected situations. Expected exceptions are foreseen by the designer who developed a specific handler (here, another protocol). Unexpected exceptions are not coded beforehand and some mechanisms allow to dynamically build a handler from a base set.

The method has been illustrated for a hotel reservation protocol. An expected exception can be the case where there is no vacancy in the hotel. The system designer usually foresees this issue and a specific handler is available in the system to deal with it. An unexpected exception can be the start of a fire that would oblige the hotel to redirect all clients to an alternative business partner. The designer might not foreseen—or enough time to foresee—such a situation. Mallya

and Singh propose to rely on an external exception handler repository to fetch a specific handler and merge it automatically with adequate system protocols. In other words, this approach elaborates on the model of Klein et al. to introduce a shared repository of protocols [51].

This approach respects the assumptions of MAS introduced in this document, as it was explicitly designed for open systems with autonomous agents. However, the work is mostly theoretical and it lacks validated results in practice. The current issues are the computational complexity of handler selection and dynamic assembly of new handlers [59]. The adoption of the architectural choice of Klein et al. is said to partially solve these issues, but complexity and scalability remain to be evaluated. The main contribution of this work is therefore the illustration of exception handling mechanisms that hold at the agent level, that take into account the case of unexpected situations, and that might be practical for a certain number of agents.

2.5.4 Stigmergic systems

Stigmergy is an interaction model where agents put marks in the environment (messages with no intended recipient) that other agents exploit to determine their next actions. Stigmergy models and allows to simulate the behavior of some social insects such as termites. One termite starts to build a nest by putting a piece of material on the ground (a mark). Other termites use this information to determine where to pile the piece they carry. Stigmergy is thus an indirect interaction model as there is no direct message passing.

Stigmergic systems are particularly robust to some types of agent exceptions such as the death or the failure of agents [71]. The robustness of these systems is mostly due to the high redundancy of agents, which reminds the choice for modularity of software architectures that could limit the impact of exceptions in sequential systems.

There is little work on stigmergic systems that discusses robustness issues, and no work on exception handling to date. Although the robustness inherent to such systems entails that no significant advance might be expected in exception handling, recent extensions of stigmergic systems to 'human stigmergy' are to be demanding for such techniques [72]. As for architectural considerations, stigmergic systems emphasize the importance of the application environment in the robustness of the system. The environment can be thought of as a 'glue' in-between agents that adequately diffuses the information ensuring system robustness.

Despite the potential of stigmergic systems, there are however not studied in further details in the scope of this document. One of the main reasons for this choice is Stigmergic systems are usually based on reactive agents, by opposition to the present focus on knowledge-based agents.

2.5.5 SaGE in the MadKit platform

Souchon et al. proposed the SaGE framework (acronym for Agent exception handling system) [96]. SaGE extends the exception handling system of Java with facilities to handle issues specific to autonomous agents in the MadKit platform [57]. In MadKit, agents hold some roles and provide services to each other according to the roles. Exceptions can occur at each of the three levels of service, agent, and role. The propagation of exceptions in search for a handler follows a predefined chain order. The possible chains share the same search order with services, agents, then roles, and finally the calling service (the propagation to role is skipped when only one agent is involved in the handling). In addition, SaGE provides a mechanism for 'concerted exception handling' to resolve errors depending on several agents. This mechanism allows to specify when agents effectively recover from some errors. Agents 'wait' until sufficient reasons are collected to react to an error at the service and role levels. Souchon et al. advance that the concerted exception model allows to avoid reactions to under-critical situations and to collect exception reports so as to evaluate a collective state.

An example of concerted exception handling in SaGE is implemented in a travel reservation system where several service providers encounter a failure. When few providers fail, limited results can be generated in a degraded mode. Too many failures compared to the number of providers trigger a specific method in the agent code to terminate properly the transaction for the reservation.

SaGE complies partially with the agent exception definition, owing to the focus on autonomous agents. However, SaGE does not scale to heterogeneous system issues as it assumes benevolent agents only. Nevertheless, SaGE brings notable instances of mechanisms for exception handling to the agent-oriented engineering community, namely the propagation that follows an agent-specific organization model (AGR, Agent-Group-Role [22]) and the concerted exceptions.

2.6 Survey conclusion

Related work on exception handling spans over research in Software engineering and Artificial intelligence. As Multi-agent systems rely on these two research domains, a number of concrete achievements can be observed, either for the theory underlying MAS or the practice of building them.

Most work do not however comply with the necessary requirements to deal with the idea of exception in MAS. Current achievements do support MAS as software entities: Programming exceptions are now well-known concepts. They do not support MAS to a sufficient extent as an *open and heterogeneous system of autonomous agents*.

The different approaches related to distributed systems, architecture, components, and earlier work in MAS identified some of the essential issues to address a full-fledged exception management system, notably the problems of concurrency in handling or the systemic dimension. They can handle to some extent with the is-

sues of openness and heterogeneity. They usually cannot cope with the assumption of autonomy.

One surprising result of this survey is that there is almost no attempt to give a clear definition of the concept of exception in MAS, especially in the work directly related to the agent research community. Key examples are explained in detail, such as the agent death, but the concept of exception remains an intuition. For this reason, the reminder of this document proposes a definition of *agent exception* and the study of an agent execution model that better addresses the semantics of exception in MAS. Although the model does not cover to full extent the issues of agent exception, it settles the foundation for future work with respect to agent autonomy.

Three

Definition of Agent Exception

The current achievements for exception management in Multi-agent systems have given the intuition that agents can encounter events that are *not* programming exceptions, while they still need to consider these events as unexpected or rare situations. This intuition leads to the concept of *agent exception* that is developed in this chapter. The starting point to analyze and determine an acceptable definition of agent exception is the original definition of programming exception.

In the era of procedural languages and object-oriented programming, the term 'exception' has acquired a specialized meaning, tightly attached to high-level programming paradigms, as illustrated by the definition of Goodenough [34, 35, 33].

> Of the conditions detected while attempting to perform some operation, exception conditions are those brought to the attention of the operation's invoker. The invoker is then permitted (or required) to respond to the condition.

When a program attempts to call an operation in its execution flow, the operation must check conditions that must hold before the actual performance can occur. In case at least a condition is not passed, the operation returns a message to the caller stating that it cannot execute due to the condition violation. This semantics was extensively discussed in the previous chapter.

This definition applies to the different elements of a MAS, namely agents, environment, and deployment context (e.g. resources such as databases) since they are all software programs. However, the characteristics of MAS and the study of related work show that this definition is not adapted to fully address agent exceptions, owing to the characteristics of openness, heterogeneity, and autonomy. The aforementioned exception definition makes the invoked operation declare unequivocally that a situation is exceptional. Such approach is not inappropriate to MAS, where *equivocal interpretation* should occur. In fact, an agent can be deemed as autonomous if it can decide by itself. The semantics of programming exceptions does not allow such decision, as illustrated in the following Fig. 3.1 and Fig. 3.2.

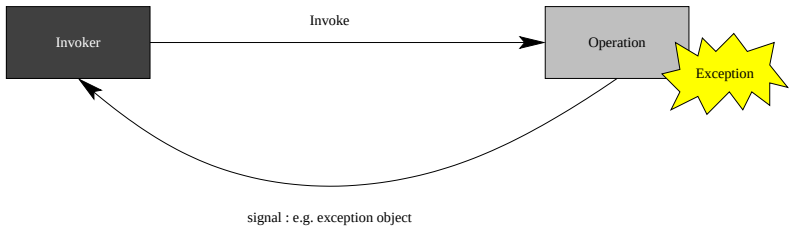


Figure 3.1: View on the semantics of programming exceptions: Exceptions are decided on the operation side

When an exception is decided on the side of the operation, the invoker has no decision to perform. The reply from the operation is for example an exception object in many object-oriented programming languages. This mechanism does not however map on the target for agent exceptions, as shown hereafter.

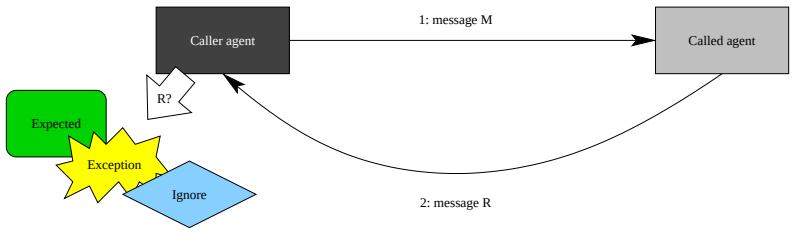


Figure 3.2: Target of the semantics of agent exceptions: Exceptions are decided by agents themselves

Autonomous agents should be able to decide whether a message sent by other agents (after a request or in the first place) is either expected, exceptional, or e.g. to ignore. This claim can also be generalized to any input received by an agent, either from peers, the environment, or elements in the deployment context [107].

3.1 Agent exception

According to the characteristics of MAS, the model of agent exception developed in this document is defined as follows. The terms used shall be understood at the granularity of agents, i.e. the syntactic unit that exceptions target is an entire agent entity and not only a block of commands in its code.

Definition: *Agent Exception*

An *agent exception* is the interpretation *by* the agent of a perceived event as unexpected.

This definition sets forth the role of the agent in the decision process of exceptions, which is relative to events that are perceived by the agent, as introduced on Fig. 3.2. When an agent receives an input, it can decide how to classify this input. The decision criteria for exception is the *expectation*. Agents are knowledge-based entities that execute protocols[1]. The activities and goals of the agent allow to formulate expectations for the future evolution of the world: Agents send messages to one another in the aim to receive certain results, which are expectations. An agent is consequently able to interpret an input as unexpected, whenever this input does not match its expectations.

The meaning of an agent exception then differs significantly from programming exceptions. When the latter is associated to an *event*, the former is associated to the *interpretation of an event*. Autonomous agents can then keep the control of themselves and decide how to process an input.

This definition provides the basis of what an agent exception is. One argument could be formulated to weaken this definition. Autonomous agents are often expected to execute in the context of an *organization*. An organization defines power relationships, already mentioned in the introduction as *social dependencies* (page 7). Such relationships are to guarantee that, despite autonomy, agents comply with the requests that are sent to them. Such settings is appropriate in closed systems where the software designer controls all parts of the system. The typical 'supervisor-worker' model does actually rely on the assumption that workers *obey* the supervisor. In open settings, individual agent designers want to keep their own agents under full control, and they want to decide how the agents respond to solicitation from external, perhaps unknown, agents. Despite power relationships between two agents, the autonomy that should be preserved leads to the aforementioned definition. Agents first decide how to process an input on their own.

This model does not contradicts the power relationships settled by organizations. Organizational power is simply thought of as an *overlay* on the autonomy of agents. Once an agent has decided whether an event is an exception—according to its interpretation—the agent can revise its decision depending on some power relationship. In other words, a worker agent can refuse to terminate on the order of a 'chief agent', if e.g. the two agents belong to different companies and collaborate in a virtual shared space.

3.2 Programming and agent exceptions

Although programming and agent exceptions are conceptually different, they pertain to the same program and they are consequently related. Programming exceptions impact the stability of the agent execution by deviating the execution flow to ex-

[1]The reader is reminded that this statement applies in the context of this document and other models of agents do exist. The approach can be seemingly adapted to other agent models, provided an appropriate representation for 'expectations' can be determined.

ception handling code and attempting to restore a consistent state. They are then activating mechanisms 'at the code level'. Agent exceptions impact the activity of the agent. The stability of the agent execution is maintained: Although the execution flow is directed to 'agent handlers', the agent remains in a consistent program state. It has to act so that its activity can continue in the context setup by the agent exception. Agent exceptions then activate mechanisms 'at the agent level'.

Definition: *Code and agent levels*

In Multi-agent systems, programming exceptions are internal conditions. They impact the *code level* of the agent. Agent exceptions are related to the activities of the agent and they impact the *agent level*.

This situation is depicted on Fig. 3.3.

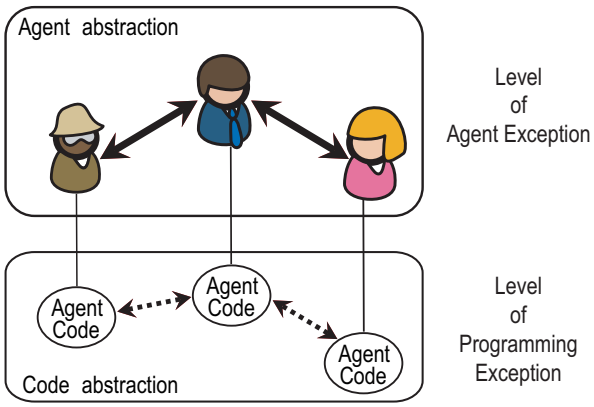


Figure 3.3: Agents and their exception levels

The aim of this section is to present the relationships between these two levels and to identify the exception spaces of agents.

3.2.1 From programming to agent exceptions.

Both types are related in a variety of cases. First, a programming exception can result in an agent exception. For example, the sudden termination of an agent due to a programming exception (e.g. `NullPointerException` in Java), has direct consequences in the agency of the system. For example, a null pointer exception usually causes the premature termination of the program. Such programming exception would then entail the 'agent death exception' [53]. When an agent dies, other agents need usually to reorganize their activities to compensate the termination of one of them, which is an exception that occurs at the agent level.

Property

Programming exceptions can breed agent exceptions.

A programming exception can also occur in an agent without generation of an agent exception. For example, the agent may have to cope with network exceptions (e.g. `IOException` in Java). A handler can sometimes deal with this problem by retrying the network connection. This exception is usually managed at the code level, so that the agent continues executing.

3.2.2 From agent to programming exceptions.

Agent exceptions do not however imply programming exceptions. In particular, agents are not terminated by the occurrence of agent exceptions. In other words, agent exceptions do not cause the code of the agent to encounter a failure.

Property

Agent exceptions do not breed programming exceptions. In particular, agent exceptions occur, while the software does not encounter any faulty situation.

The reason for this property is that agent exceptions are identified in incoming events by an individual evaluation process. The event can be considered as an agent exception, whereas the code is properly executed and no programming exception is signaled. The agent continues its execution either processing the exception or ignoring the event and moving to the next execution cycle. In this process, the internal state of the agent and its low-level contents follow a normal flow, without having the agent exception causing any programming exception. In particular, the call stack of the agent runtime is not unwound due to the agent exception. The agent exception pertains to higher level units than the call stack, e.g. agent knowledge and acquaintance network.

The two aforementioned properties allow to identify a unilateral relationship between the two types of exceptions, as depicted on Fig. 3.4.

This figure represents the relationship between the spaces of exceptions that can be designed for a MAS. As for all existing types of exceptions, programming and agent exceptions differ but are related as aforementioned. The occurrence of programming exceptions can breed in some situations an agent exception (black arrow), whereas the contrary is not possible.

3.3 Exception space in Multi-agent systems

The relationship between programming and agent exceptions provides a basic classification of the exceptional situations that an agent can encounter. Further studies

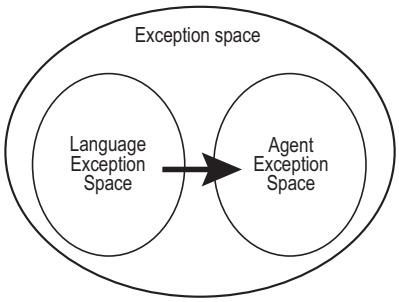


Figure 3.4: Relational mapping in an abstract exception space: Programming exceptions can breed agent exceptions, but not conversely

allow to refine the space of agent exceptions as shown in the following table. The aim of this section is to distinguish different classes of exceptions to facilitate their study and to classify the types of handlers that can be created.

		Known	Unknown
Agent Level	Coordinated	ACK	ACU
	Standalone	ASK	ASU
Code Level		CK	CU

Table 3.1: Exception space of agents: 6 classes of exception

The knowledge dimension. First of all, exceptions are usually either *known* or *unknown*[2] by the program. An exception is known whenever the program has access to a handler to manage it; otherwise, the exception is unknown. At the code level, unknown exceptions usually cause a premature termination of the program, since it cannot handle the situation and might harm the operating system or hardware low-level components. At the agent level, unknown exceptions mean the agent does not know how to react to an event given its current activities. The agent is however in a consistent state and it can decide according to its capabilities. Simple agents would just ignore the event (in the same way objects answer `doesNotUnderstand:` in the Squeak implementation of Smalltalk [97]), while complex reasoning agents would exploit the situation, such as KGP agents [48].

The scope dimension. The agent level is refined according to the scope of the agent exception. Two scopes are distinguished, namely standalone and coordinated. When an agent considers it can handle an exception without additional

[2]Related work also exploit this distinction. They however use the terminology 'expected' and 'unexpected' [59]. Given that exceptions are usually considered as unexpected situations, this document exploits the 'known' quality instead.

interaction with other agents, the exception is classified as *standalone*. When the agent considers the handling requires to coordinate with other agents, the exception is then deemed as *coordinated*. An example of standalone exception occurs in negotiation protocols, such as the CNet introduced in the case study (page 13). If a client receives an extraordinary offer, such as less than 10% of the target price offered by the client, the situation can be considered as a standalone exception that should be handled rapidly. The client can update its state so that this offer will win the call-for-proposal. Then the client can complete the run of the protocol to accept formally the offer and refuse others: No extra interaction was required to handle this particular situation. The death of the client can also be another kind of standalone exception depending on the handling strategy. Once the exception detected, providers can individually stop the corresponding activity. A simple coordinated exception can be the `DelayAnnouncement` exception introduced for the case study. A provider announces a delay to the client, who can react by granting a time extension to all providers.

Handler classification. The exception space serves to describe the exception types and to classify the handlers that apply. For example, the agent death is an agent exception that can be considered as known, since it is a basic case that can be assumed by default due to the amount of past research [101, 51, 53, 66]. The agent death is however *either addressed in a standalone or coordinated way*, depending on the type of handler that is provided to the agent to manage the case.

The classification serves two purposes. It helps to guide designers in creating handlers or techniques to develop them at runtime. Depending on the target application of the MAS, some types of handlers are necessary and others are superfluous. Handlers in the unknown category require specific techniques to search or generate them, and it can be too costly process for certain applications. The second purpose is to provide the agents with a decision criteria. Depending on the exception, a certain type of handler is searched.

Classes of handlers are defined by the acronyms in Tab. 3.3. ASK thus refers to handlers for Agent-level Standalone Known exceptions and CU refers to handlers for Code-level Unknown exceptions. In the remainder of this document, the acronyms will be used to name the handler classes.

Ordering preference. As exceptions can be either handled in a standalone or coordinated way, agents can face dilemma in deciding when a handler of each class is available. Coordination is usually an expensive matter in distributed applications and it can overwhelm the advantage of distribution by replacing the computational cost into a communication cost. For this reason, a rationale choice for agents is to prefer handlers that manage exceptions in a standalone way over others.

The possible high complexity of interactions in MAS emphasizes this ordering preference. MAS are expected to be structured in multiple organizations and to be regulated at runtime, e.g. in electronic institutions [20, 105]. These structures

produce complex social relationships among agents that may constrain the handling procedures. Consequently, it is usually significantly less expensive to attempt first a standalone resolution of an exception whenever a handler is available.

3.4 Revisiting the terminology on exception management

The semantics of agent exceptions differs significantly from the one for programming exceptions. For this reason, the vocabulary used in programming languages does not always keep its original meaning. The purpose of this section is to revisit the usual vocabulary exploited in exception management and provide definitions in the context of MAS.

Exception diagnosis (or detection) refers to mechanisms to evaluate perceived events and detect exceptions. Usual programming languages name similar mechanisms as resolution (in the case of Distributed computing).

Exception signaling does not seem to need an equivalent in agent exceptions. Indeed, signaling an exception means traditionally that an operation informs its invoker that an exception occurred. The flow of controlled is reversed back to the invoker. In MAS, the exceptions are detected by the agents, and the need to reverse the control flow disappears, as the agent continues its execution. Similar reasons pertain to exception raising, i.e. exceptions implicitly declared by the software runtime environment.

Exception propagation is the mechanism that describes how agents deal with exception situations they are unable to manage. In such case, an agent can try to find a peer agent for help. The term propagation is used to express that an exception is turned into a message (e.g. a call for support) and propagated to peers that may help. This propagation is from the point of view of the sender. For other agents, this propagation is just an event that may be evaluated as an exception. This expression then differs significantly from programming exceptions, where it means 'passing' the exception along the call stack of the process.

Exception transforming is a technique to change the type of an exception along the handling procedure when it is necessary. In distributed computing, transformations are used to find a common exception type when several software components detect an exception concurrently [112, 66]. In agent systems, the transformation mechanism is done by each agent that evaluates an event as exceptional. The reason for the difference is the loose coupling between agents. Techniques from distributed computing actually assume the close collaboration among processes, which is not always possible in open and heterogeneous systems.

Termination refers in usual systems to the end of a program caused by an exception condition ('abnormal termination'). Agent exceptions cannot cause

a termination of a MAS due to the loose coupling among agents and their autonomy. Agents are free to choose the consequence of an event (including terminating), and their choices are individual, so that the termination of an agent does not imply the termination of any other.

Resumption is usually defined as the continuation of a program execution after the handling of an exception. In agent systems, the definition is the same with different underlying mechanisms, since resumption then concerns the activities of agents.

Exception handling is the actual processing of an exceptional situation by an agent. Handling is the execution of specific tasks, while the execution of other activities of the agent are either unmodified (the exception case is ignored and the execution continues) or suspended (with subsequent termination or resumption).

Exception management refers to all activities involved in the management of exceptions by agents. It encompasses all the previous mechanisms.

In the programming language literature, the aforementioned terms can have formal models of the underlying mechanisms. This work remains to be done for Agent-oriented computing. Besides, candidate mechanisms are not necessarily language constructs. Agent exceptions are at the agent level and other 'forms' of mechanisms seem more appropriate. For example, propagation and transformation seem better served by architectural or algorithmic forms than a language construct.

3.5 Conclusion

Exception management in MAS shares the meaning of exception handling in programming languages. That is, the design of MAS must deal with the occurrences of programming exceptions, as done in usual software engineering. In addition, exception management in MAS refers to *agent exceptions*, a class of exception that differs from programming exceptions, as presented in this chapter. One conclusion of the introduction of the concept of agent exception is that designers must cope with an additional class of issues. As a consequence, the tasks of the designer becomes heavier and more error-prone, so appropriate modeling and support for agent exceptions is desired. The remainder of this document aims at providing such support. The approach on exception management will focus on modeling exceptions relative to interaction protocols, since engineering agent systems is mainly concerned with agents that coordinate and execute according to interaction protocols.

In the remainder of this document, the term exception will refer to agent exception when there is no ambiguity with the concept of programming exception.

Four

Agent Execution Model and Architecture

The definition of agent exception and related work have consequences on the execution and architecture of agents. Examples of well-known agent architectures are the Subsumption from Brooks [7], instances of the BDI model, such as in the Jadex and Jason frameworks [45, 46, 42], or the KGP model of agency [48, 98]. However, these architectures have two shortcomings as for the question of agent exceptions. They do not set forth *explicit* exception management facilities in the execution model of the agent, and they do not distinguish the case of agent exceptions. Exceptions are usually dealt with as programming exceptions and rely on the facilities provided by the underlying languages, such as Java or Prolog. *The management of agent exceptions requires however to deal with the assumptions in MAS, and Software engineering 'good practice' invites to separate explicitly the mechanisms for the application logic form the mechanisms for exception management.*

The aim of this chapter is to introduce an execution model of agent that integrates exception management in such a way that the aforementioned separation of mechanisms is realized. An agent software architecture is also proposed to better implement the execution model, still maintaining the separation of mechanisms. Although the execution model is straightforward to implement, the architecture appeared as necessary to ease the engineering of agents and improve the performance of the model based on engineering principles.

4.1 Agent Execution Model

Agents usually follow a cyclic execution model, classically the perception-reasoning-action loop [86, Chapter 2]. The model presented here adopts the same cycle, expanding the perception and action stages to appropriately setup the reasoning stage in case of exceptions, with respect to the agent autonomy assumption. This section first describes the structure of protocols, handlers, and knowledge of

the agents and then the execution model. Illustrations are given in the context of the case study presented in the introduction, page 11.

4.1.1 Model of Protocols and Handlers

4.1.1.1 Structure of messages

First, we express ACL messages with the pattern:

$$msg = (id, from, to, perform, content, time) \tag{4.1}$$

In a message msg, id identifies the protocol it belongs to, $from$ and to are the sender and receiver, $perform$ is the performative, $content$ is the message content, and $time$ is the time-stamp of the reception. The intent of this formal representation is to have a sufficient and compact representation of agent communications in the frame of this paper. This representation can be replaced by FIPA-ACL, if required [23]. When either of the message parameter is the underscore character '_', it means the parameter is undefined and can be any value.

4.1.1.2 Structure of protocols and handlers

Protocols and Handlers as trees. AUML and related work on, e.g. commitment protocols, have represented protocols and handlers as either sequence diagrams [2] or graphs [58]. We chose to express protocols and handlers in terms of graphs in order to establish a formal representation. They are represented as directed rooted trees, where the root is the first message sent, and the graph is structured according to the relation $\mathcal{R}$, defined as follows. For any directed rooted tree $\mathcal{T}$, we note $\mathcal{M}$ the set of its edges, where the edges correspond to actions like sending messages in protocols and handlers. We also note $\mathcal{L}$ the set of leaves in $\mathcal{T}$ ($\mathcal{L} \subset \mathcal{T}$).

$\mathcal{R}$ is a transitive, asymmetric, and irreflexive binary relation.
$\mathcal{T}$ verifies the following structural properties.
(1) $\forall m \in \mathcal{M} \setminus \mathcal{L}, \exists m' \in \mathcal{M},\ m \mathcal{R} m'$
(2) $\forall m \in \mathcal{M} \setminus \mathcal{L}, \text{succ}_{\mathcal{T}}(m) = \{msg \mid m \mathcal{R} msg\}$
(3) $\forall m \in \mathcal{M} \setminus \{\text{root}\}, \exists m' \in \mathcal{M},\ m' \mathcal{R} m$

Property (1) states that each message sending has a successor but leafs. We note $\text{succ}_{\mathcal{T}}(m)$ the set of successors for a given edge of $\mathcal{T}$ in property (2). Property (3) expresses that each message sending has a predecessor, but the root.

Some protocols may contain cycles in their specifications. In those cases, the tree representation applies by 'unfolding' the loops along a branch of the tree. Such unfolding operation is common with graphs, for example with Petri nets, e.g. [61].

Protocol representation. The following algebra on message sending serves to describe protocols. The syntax relies on the set of messages $\mathcal{M}$ and the structure of the protocol complies with the previous tree model.

$$proto ::= end \vee msg \vee proto, proto \vee proto* \vee proto \mid proto \tag{4.2}$$

The name of the message *msg* is the action of sending the message. The special action *end* is a termination message that states the end of a protocol: $\mathcal{L} = \{end\}$. *msg* is the sending of a message as previously defined. $proto, proto$ means the agent executes in sequence two protocols. $proto*$ means a message can be sent optionally more than once in the protocol. It is a convenience formula that expresses the sequence of several protocols (use in sequence of the previous operator). $proto \mid proto$ means the agent can choose the protocol execution it wants to follow. The CNet exploited in the case study is therefore the following series (based on the FIPA version in [27]):

$$\begin{aligned} cnet = (&cfp,\\ &(refuse*, end*) \mid propose*,\\ &(rejectProposal*, end*) \mid acceptProposal*,\\ &(failure, end) \mid result,\\ &end) \end{aligned}$$

Handler representation. Handlers differ from protocols in that the series of a handler can contain either message sending or other types of actions internal to the agent, such as the update of knowledge or operations on protocols (e.g. suspend, resume, etc.). Internal actions are considered as 'silent transitions' similarly to τ in the π-calculus [67], so that the same notation is adopted and extends the algebra for protocols presented in formula (4.2). The set of these actions is $\mathcal{M} \bigcup \{\tau\}$, and simply noted as $\mathcal{M}$ for short. The algebra for handlers is therefore:

$$\begin{aligned} handler ::= \; & end_h \vee end_p \vee \tau(?) \vee msg \vee handler, handler \vee \\ & handler* \vee handler \mid handler \end{aligned} \tag{4.3}$$

The semantics coincide with the protocol algebra on the common operators. The handler algebra however deals with actions that are either the end_h message to terminate coordinated handling, the end_p message to terminate a protocol, an internal action $\tau(?)$, the sending of a message *msg*, a succession, or the branching on handlers. The formula $\tau(?)$ is a convenience notation where the question mark should be replaced by an application-dependent description of the corresponding internal action, either add or remove data from the agent knowledge base (with update as a successive application of remove then add). The types and effects of internal actions are described in the following section 4.1.3.

Branches of the handler do not necessarily end with the end_p message, which indicates the end of the protocol interrupted to run the handler. The message can be sent however when the handling procedure requires such action. All leaves of the tree end with the end_h message to terminate the handler: $\mathcal{L} = \{end_h\}$. We describe hereafter three example handlers for the same exception type. An ACK handler for the `DelayAnnouncement` exception of the case study can be the following:

$$extendTime = (\tau(\text{Update internal timeout}),\ inform(\text{"}newtimeout\text{"})*)$$

The handler leads the client to update the internal value of the timeout for the corresponding protocol, then to send a message to all providers about the new deadline for replies. An ASK version of $extendTime$ would be the same formula without sending the inform message, i.e. the formula contains no message sending.

Another simple handler for the same delay request is to refuse in any way. The handler is expressed as follows.

$$h = (\tau(\text{“Remove expectations for the sender in this protocol”}),\ \text{refuse})$$

The delay request is refused with the corresponding message. In addition, the agent updates its knowledge base by removing the requester from the protocol participant list, since the agent decided not to interact anymore with the requester on this protocol instance. A more flexible version of the handler is as follows, which asks for reactions depending on the delay value.

$$h = (\tau(\text{“Check delay} \leq \text{1s”}),\ \text{accept}) \mid (\tau(\text{“Check delay} > \text{1s”}),\ \text{refuse})$$

If the delay lasts for less than a second, the delay is accepted, whereas it is refused otherwise.

Unfold representation. The above representations of protocols and handlers are introduced for more compactness of formulas, but they abstract the details. According to the form of a message and the algebras, protocols and handlers are actually longer formula where all necessary details are given. The unfold representation can be used when details must explicitly appear. For example, the $extendTime$ handler is represented as follows:

$$\begin{aligned} extendTime = (&\tau(\text{Update timeout}),\\ &(id, myself, \{\text{provider list}\}, inform, deadline = T, \_)) \end{aligned}$$

The message $inform(\text{“}newtimeout\text{”})$ is unfold to reveal its parameters. Messages that are sent several time (noted with $msg*$) appear with a list of recipients. The time parameter is unknown as it is intended to contain the reception time-stamp.

4.1.1.3 Semantics of the protocol and handler execution

The execution of agents follows the semantics hereafter. The similar syntax of protocols and handlers allows to construct a common execution mechanism. First, we define sets of messages, protocols, handlers, and neutral elements, i.e. a protocol and a handler that do nothing.

$$\begin{aligned} &\mathcal{M},\ \text{Set of messages}\\ &\mathcal{E},\ \text{Set of execution histories}\\ &O_{\mathcal{E}} \in \mathcal{E},\ \text{Empty execution} \end{aligned}$$

The following mechanism 'execute' describes the evolution of the protocol and handler executions by the agent.

$$
\begin{array}{lrcl}
\text{execute:} & \mathcal{M} \times \mathcal{E} \times \mathcal{E} & \rightarrow & \mathcal{E} \times \mathcal{E} \\
& (m, E_p, O_\mathcal{E}) & \mapsto & \begin{cases} (E_p \bigcup \{m\}, O_\mathcal{E}) & \text{if } m \neq end \\ (O_\mathcal{E}, O_\mathcal{E}) & \text{if } m = end \end{cases} \\
& (m, E_p, E_h) & \mapsto & \begin{cases} (E_p, E_h \bigcup \{m\}) & \text{if } m \notin \{end_p, end_h\} \\ (E_p, O_\mathcal{E}) & \text{if } m = end_h \\ (O_\mathcal{E}, E_h \bigcup \{m\}) & \text{if } m = end_p \end{cases}
\end{array}
$$

The execution evolves depending on the type of message received by the agent and the state of protocol and handlers it is running. The first case $(m, E_p, O_\mathcal{E})$ expresses the execution of a protocol p. The history of execution grows while messages are processed (reception and sending), and the execution stops whenever end is received, which empties the history for the completed protocol. In the second case (m, E_p, E_h), a handler is executed by the agent, so that the handler execution supersedes the protocol one. If the message is end_p, the protocol is stopped and the handler execution continues. If the message is end_h, the handler execution has successfully completed and the protocol execution is restarted.

4.1.2 Structure of Knowledge

Agents maintain some data structures in order to process their inputs and distinguish exceptional conditions from 'normal' ones. The contents of these data structures is expressed in first-order predicate logic, and we assume all identifiers are unique.

The key knowledge that agents manage in the detection and handling of exceptions is the *expectation* [10, 9]. In the present case, expectations are defined relatively to protocol and handler sequences: At any step of a sequence enactment, the expectations of the agent are the next possible steps in the sequence. Such model of expectation and appropriate comparison mechanisms allow to detect exceptional situations and trigger handling procedures, as presented in this section.

4.1.2.1 Tabular knowledge

Agents maintain specific knowledge structures that we represent as tables for readability. They first have a relevance table $relTab$ that gathers 'filters' for the agent input, as shown in table 4.1. Filters are patterns of acceptable messages. Messages that do not comply with the filters are discarded. Filters allow to ignore all messages that may yield exceptional conditions but that are not relevant to the agent in any case. Filtering out these messages before any further process is necessary in dynamic and open environments due to the computational cost [55].
In the case study, an agent considers already running CNet $cnet_1$ and $cnet_2$ as relevant (first two entries), in addition to any message from agent ag_1 (the underscore '_' means 'any value').

id	from	to	perf	content	time
$cnet_1$	_	_	_	_	_
$cnet_2$	_	_	_	_	_
_	ag_1	_	_	_	_

Table 4.1: Sample Relevance Table $relTab$

The second table $expecTab$ stores the expectations of the agent, as presented in table 4.2. Expectations are patterns of messages that the agent is waiting for with regards to the sequence of messages in an interaction protocol. Expectations allow the agent to distinguish *expected* situations from exceptional conditions, i.e. *unexpected* cases, hence the name of the .

id	from	to	perf	content	time
$cnet_1$	ag_1	$self$	$inform$	$bid(s_1, \_)$	$t < T_{bid}$
$cnet_2$	_	_	_	_	$t < T_{lim}$

Table 4.2: Sample Expectation Table $expecTab$

Table 4.2 shows an agent in the case study that expects a bid from ag_1 relative to the running protocol $cnet_1$ before a certain date T_{bid} is reached (first entry of the table). Failing to receive such a message on time will be considered as an agent exception. The second entry is more general: The agent expects to be informed about any message relative to $cnet_2$, for example as a referee.

Table 4.3 shows a execution table of an agent, where each running and suspended protocol and handlers involving the agent is a 3-tuple entry.

id	state	dependency
$cnet_1$	$running$	nil
$cnet_2$	$suspended_4$	$hand_1$
$hand_1$	$running$	nil

Table 4.3: Sample Execution Table $exeTab$

In this sample table, the protocol $cnet_1$ is running, whereas protocol $cnet_2$ is suspended in the fourth step of its execution. $cnet_2$ waits for the completion of the running handler $hand_1$.

Finally, agents maintain a handler table as in table 4.4, so that to relate unexpected events with known handlers. Each handler is related to one or more conditions that describe the kind of events where the handler is applicable. A condition can also lead to several handlers that the agent can choose from at runtime.

condition	handler
$(\_, \_, \_, inform, delay(\_, \_), \_)$	DelayAnnouncement
$(\_, \_, self, \_, \_, now > time)$	DelayedAnswer
$(id \notin exeTab, \_, \_, \_, \_, \_)$	UnexpectedProtocol

Table 4.4: Sample Handler Table $handTab$

In the case study, an agent detects a delay announcement whenever a message matches the pattern in the first entry, i.e. an $inform$ message (as for the FIPA-ACL like semantics [23]) with a predicate formula that announces a delay. When such message is encountered and deemed as unexpected, the corresponding *DelayAnnouncement* handler is selected.

In the remainder of this section, we will use a convenience macro-formula about these tables, formally defined as:

$$match((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \equiv (\forall i,\ a_i = b_i)$$

In other words, two tuples 'match' whenever their values are exactly the same.

4.1.3 Execution model

The overall execution model of agents is presented in Fig. 4.1, with four levels. We describe them in turn in the following. Descriptions rely on algorithms for the domain-independent parts, whereas the descriptions are restricted to the case of the case study for domain-dependent parts.

4.1.3.1 Top-level execution

Message reception, relevance filtering, & expectation matching. The top-level boxes (in white on the figure) are the fundamental stages of the execution model. Input messages are collected by the agent in the *Message Reception* box. They are forwarded to the *Relevance Filter* to filter out messages that do not matter for the agent according to the relevance table $relTab$, depending on its autonomous choice. Relevant messages are then compared to the agent expectations in the *Expectation Matching* box, depending on the expectation table $expecTab$ of the agent. Alg. 4.1 describes the functioning of this stage.
The message is compared to each entry of $expecTab$ until either a match is found, or the table is entirely read. If a match is found the output is an expected message '$expMsg \leftarrow msg$' and 'nil' for the $unexpMsg$. If no match is found, the contrary assignment is performed. In the former case, the *Decision Process* is then activated, whereas it is the *Handler Selection* in the latter.

Decision process. As for the top-level boxes, expected messages are forwarded to the *Decision Process*, which is the reasoning part of the agent. The message

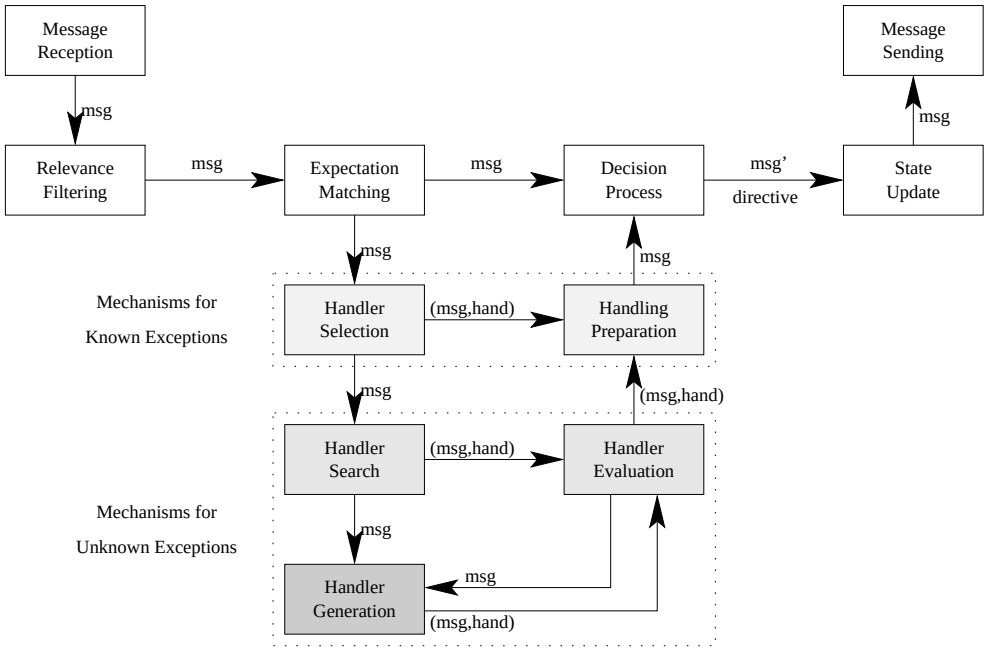


Figure 4.1: Execution model of an agent with incremental exception handling mechanisms

Input: $msg, expecTab$,
Output: $expMsg, unexpMsg$
$expMsg \leftarrow nil$, $unexpMsg \leftarrow nil$;
foreach $tuple \in expecTab$ **do**
  if $match(tuple, msg)$ **then**
    $expMsg \leftarrow msg$;
    Break for loop;
  end
end
if $expMsg = nil$ **then**
  $unexpMsg \leftarrow msg$;
end

Algorithm 4.1: Expectation matching

is there processed to determine the next action of the agent, if any, as shown in Alg. 4.2. In addition to this algorithm, it should be noted that the Decision process box executes continuously and it does not require an input message to trigger an output. This execution is not represented in the algorithm as it does not participate in the exception handling cycle. It is however important as it is the 'proactive' part of the agent, necessary for the agent to initiate activities. In that sense, proactive agents differ from Mealy machines.

```
Input: msg, exeTab,
Output: msg′, directive,
Input/Output: K
msg′ ← nil;
foreach tuple ∈ exeTab do
    if match(tuple, (msg.id, running, _)) then
        msg′ ← decide(msg, K);
        directive ← generateDirectives(msg′, exeTab);
    end
end
```

Algorithm 4.2: Decision process

Depending on the current advancement in the protocol concerned with the incoming message, the agent takes some decisions in *decide* and generates relevance and expectation directives in *generateDirectives*, both domain-dependent functionalities of the agent that exploit the knowledge of the agent $\mathcal{K}$. The *generateDirectives* is however endowed with a domain-independent mechanism to produce expectation and relevance filters. This mechanism produces directives about the expected and relevant messages awaited by the agent according to the running protocols and handlers. Alg. 4.3 details the basic algorithm. This algorithm is not concerned with starting a protocol or with additional updates that can be done afterward in a domain-dependent manner.

Alg. 4.3 processes domain-independent data only, namely the different knowledge structures for managing the agent execution and the type of messages processed by the agent.

The algorithm exploits the message msg' produced by the agent decision process to generate the directives used in the next state update. The outer loop of the algorithm reviews in turn each tree of the agent execution table (protocols and handlers). If msg' is the root of the current tree, it means the agent has proactively initiated a protocol (*msg* is void). Two directives are produced to update the relevance and expectation tables with information about the new tree. If msg' terminates a tree with either *end* or end_h, relevance and expectation tuples for the corresponding tree are removed from the tables. The case of end_p is particular, since it occurs during the execution of a handler to terminate the attached protocol. The *exeTab* contains a *dependency* column that serves to retrieve the reference of the protocol to terminate, so that two directives are produced to remove the related information in the tables. All other cases in the enactment of a tree require removing the expectation rules that are outdated and the addition of the next expectations. The relevance table do not need to be updated at this stage, since the corresponding protocol is running and still useful to the agent.

The decision eventually outputs a message msg', which can be either a message according to the next steps of the protocol, or a 'null action' when the agents decides

Input: $msg', exeTab$,
Output: $directives$,
Input/Output: $relTab, expecTab$
$directives \leftarrow \emptyset$;
foreach $tree \in exeTab$ *where* $tree.id = msg.id$ **do**
  if msg' *is root of* $tree$ **then**
    $directives \leftarrow \{relTab \leftarrow relTab \bigcup \{(tree.id, \_, \_, \_, \_, \_)\}\}$;
    $directives \leftarrow \{expecTab \leftarrow expecTab \bigcup \mathrm{succ}_{tree}(msg')\}$;
  else if $msg' \in \{end, end_h\}$ **then**
    $directives \leftarrow \{relTab \leftarrow relTab \setminus \{t \mid t.id = tree.id\}\}$;
    $directives \leftarrow \{expecTab \leftarrow expecTab \setminus \{t \mid t.id = tree.id\}\}$;
  else if $msg' \in \{end_p\}$ **then**
    $directives \leftarrow$
      $\{relTab \leftarrow relTab \setminus \{t \mid t.dependency = tree.id\}\}$;
    $directives \leftarrow$
      $\{expecTab \leftarrow expecTab \setminus \{t \mid t.dependency = tree.id\}\}$;
  else
    $directives \leftarrow \{expecTab \leftarrow expecTab \setminus \{t \mid t.id = tree.id\}\}$;
    $directives \leftarrow \{expecTab \leftarrow expecTab \bigcup \mathrm{succ}_{tree}(msg')\}$;
  end
end

Algorithm 4.3: Generation of directives in *generateDirectives*

to ignore the input (Alg. 4.3 is then not executed). The directives serve in the next stage *State Update* to update the relevance and expectation filters for the future cycles, and to commit the optional action in the environment in *Message Sending*.

State update. Alg. 4.4 describes the update procedure for the knowledge of the agent. The tables are updated in turn. The order of update does not matter in the model[1].

Input: $directive$,
Input/Output: $msg, relTab, expecTab, exeTab$
$update(exeTab, msg, directive)$;
$update(expecTab, msg, directive)$;
$update(relTab, msg, directive)$;

Algorithm 4.4: State update

The *update* function is domain-independent as it merely applies the directives on the tabular knowledge of the agent, as indicated in Alg. 4.3.

[1]It was however convenient in practice to enforce the order to perform some consistency checks on the tables.

4.1.3.2 Exception handling execution

The agent detects exceptional conditions whenever an expectation is not met at the *Expectation matching* stage. The execution flows of the agent then continues through the deeper levels of the model.

Handler selection. The second level is to deal with *known exceptions*, i.e. when the agent owns a handler in $handTab$ that applies to the detected exception. Unexpected messages are sent to the *Handler Selection* box, where a handling procedure is searched for as done in Alg. 4.5.

```
Input: expecTab, handTab,
Output: hand,
Input/Output: msg
Require msg ≠ nil;
hand ← nil;
foreach tuple ∈ handTab do
    if match(msg, tuple.condition) then
        hand ← preferred(hand, tuple.handler) ;
    end
end
```

Algorithm 4.5: Handler selection

Alg. 4.5 searches the handler table $handTab$ of the agent for an appropriate handler. If an entry of the table has a condition that matches the message, a handler is found and returned by the algorithm. In case several handlers are found, the $preferred$ function allows to determine which handler is preferred to the agent, depending on its configuration and context. The $preferred$ function is consequently a domain-dependent part of the algorithm. Preference functions typically adopt a metrics to evaluate handlers. For example, the case study can accept a preference function that prefers ASK handlers over ACK ones for performance issues. Another criteria can be the complexity of the handler, based on e.g. the length of its sequence.

Handling preparation. If a handler is found, it is forwarded with the unexpected message to the *Handling Preparation* stage detailed in Alg. 4.6. The preparation consists in suspending the protocol impacted by the incoming message, starting the handler as an activity of the agent, and specifying that the suspended protocol must be evaluated when the handler completes by creating a dependency of the protocol for the handler. The agent can then decide whether to resume the suspended protocol where it was interrupted or to terminate it. The preparation concludes with the sending of the message to the Decision process, ready for dealing with the exception, owing to the prepared handler.

Input: $hand$,
Input/Output: $msg, exeTab$
foreach $tuple \in exeTab$ **do**
  if $match(tuple, (msg.id, \_, \_))$ **then**
    $msg_i \leftarrow$ Get the expectation before handling for $msg.id$;
    $exeTab \leftarrow exeTab \setminus \{tuple\}$;
    $exeTab \leftarrow exeTab \cup \{(tuple.id, suspended_{msg_i}, hand)\}$;
    Break for loop;
  end
end
$exeTab \leftarrow exeTab \cup \{(hand, running, nil)\}$;

Algorithm 4.6: Handling preparation

Handler search & evaluation. If no handler is found in the selection stage, the agent encounters an *unknown exception*, i.e. the agent does not own any corresponding handler. The agent can then try a *Handler Search* by interacting either with other agents in the system or with a handler repository. A query is sent to a collaborative agent or such repository to attempt finding a handler (see [51, 82] for original approaches). The success of the search produces a handler that is forwarded to the *Handler Evaluation* for checking the adequacy to the problem at hand, maintaining the agent autonomy relative to this external handling code, and linking the exception type to the handler in the handling table if the agent keeps this handler. In general, the evaluation process is a complex issue and we adopt a simplified approach for the presentation in this section: We assume that if the handler leads to a state where the interrupted protocol can resume execution, then it is acceptable.

$$\begin{array}{l} \text{Definition of handler acceptability: } h \text{ acceptable i.f.f. } h_n = p_{interr} \\ \quad \text{With handler:} \quad h = (h_i)_{i \leq n}, \\ \quad \text{And protocol:} \quad p = (p_i)_{i \leq m}, \text{ interrupted at step } p_{interr}, \\ \quad \text{And:} \quad end_p \equiv end \end{array} \tag{4.4}$$

In other words, the agent 'trusts' external code whenever it leads the execution to the previously desired state before the occurrence of the exception. This simple check does not allow however to guarantee that any step of the handler is acceptable for the agent. Such general mechanism would depend on the application domain.

Handler generation. In the final case were no handler can be found (or when the evaluation is not satisfactory), the agent then attempts a *Handler Generation*. In our approach, this generation necessarily produces a default handler if no better solution is available. This default generation is essential for the continuity of the execution model, to ensure the model does not stop in such process. The default handler of the execution model is to 'ignore' messages a fixed number of times

before considering the corresponding protocol has failed. The generated handler is parameterized with the expected message and put in the protocol table at the Handler preparation stage. For instance, the handler h_{gen} generated to wait for the message *msg within three execution cycles concerning the protocol p* is defined as:

$$\begin{aligned} h_{gen} = ((msg, end_h)|(msg, \tau(\text{Ignore})),\\ (msg, end_h)|(msg, \tau(\text{Ignore}),\\ msg|(msg, \tau(\text{Update protocol information}), end_p, end_h)) \end{aligned}$$

The agent then expects to receive the msg in one of the three steps of the handler, and then returns to the protocol with end_h. At each step, the agent ignores the message if it is not conforming with its expectations. Beyond the reception of three non-conforming messages, an internal action update the protocol state, and a protocol cancellation message (end_p) is sent to all the participants.

4.1.4 Complexity analysis

Given the algorithms presented in this section, the purpose of the complexity evaluation is to measure the overhead computational cost of the exception handling mechanism. We compare the cost of the expected execution model to the exceptional one to this end. Practical evaluation based on this analysis are conducted in chapter 5 on the validation of the model.

Notations. In usual agent execution models, the cost is concentrated in the Decision process box, without generation of directives. This cost is application-dependent and is noted N_{DP}. The other application-dependent algorithms are for Handler evaluation and Handler generation, with respective cost N_{Eval} and N_{Gen}. Let us also note n_{pro} the size of the protocol table $exeTab$, representing the number of protocols or handler exploited by the agent (state is running or suspended), and n_k the number of handlers known to the agent.

Basic complexities. In the overall agent execution model, the complexity of each box depends on the number of directives generated by the decision process. In fact, the number of directives entails the production of expectations and relevance, which directly influence the cost of the filtering, matching, and state update. A typical execution cycle generates up to 3 directives per protocol in the present approach (due to Alg. 4.3), which entails a complexity of $O(n_{pro}$. Most algorithms have then a complexity related to the size of the protocol table $exeTab$, i.e. n_{pro}. We have the following costs for each algorithm.

- $O(n_{pro})$ for $generateDirectives$
- $O(n_{pro})$ for Relevance Filter and Expectation matching
- $max(N_{DP}, O(n_{pro}))$ for Decision Process

- $O(n_{pro})$ for State Update
- $O(n_k)$ for Handler Selection
- $O(n_{pro})$ for Handler Preparation

As for the Handler search part, the computational cost is not significant compared to others. The reason is that the search is a request to other agents or handler repositories. The communication cost is however increased, but it does not participate in the computational one we are evaluating here. The following table 4.5 compiles the different overhead costs depending on the execution type.

Execution type	**Cost**
No handling	N_{DP}
With handling	$N_{base} = max(N_{DP}, O(n_{pro}))$
Overhead cost over N_{base}	
Known exception	$max(O(n_k), O(n_{pro}))$
Unknown exception, Search & Evaluation	$max(N_{Eval}, O(n_k), O(n_{pro}))$
Unknown exception Search, Generation, Evaluation	$max(N_{Eval}, N_{Gen}, O(n_k), O(n_{pro}))$

Table 4.5: Cost table depending on the execution type

The complexity of the agent execution model increases by one order with the only introduction of the top-level elements of the handling mechanisms (the white boxes on Fig. 4.1). If we assume that in practice agents are expected to run a low number of protocols simultaneously, a cost of $O(n_{pro}^2)$ (assuming $O(n_{pro}) \sim N_{DP}$) can then be reasonable compared with an agent without exception management system. It is however prohibitive for 'heavy-weight agents', for which some other kinds of fault-tolerance mechanisms can be considered.

The complexity of cycles where an exception is detected depends on the necessary depth to handle the case. In addition to the number of protocols, the number of handlers that the agent knows is a costly factor as soon as the handler table must be searched. An interesting result of this complexity table is that it might be better to delegate the search of handlers as long as the evaluation algorithm is less expensive than the selection algorithm, i.e. $N_{Eval} < O(n_k)$. However, we think that a robust evaluation is costly, especially to guarantee the handlers are *entirely* acceptable (in this analysis, the handler evaluation only guarantees the consistency of the result, see page 60 with formula (4.4)).

The generation adds another parameter in the complexity evaluation. In most cases, the generation of a default handler to ignore the unexpected event and lead the agent to a consistent state should not be costly to produce and evaluate. The complexity depends however on the level of generation desired for the agent.

4.2 Agent architecture

The architecture of the agent aims at setting forth the architectural elements that corresponds to the execution model of agents presented in the previous section. The architecture is abstract in the sense that it defines the elements required to design agents with exception management capabilities. As for the literature on software architecture, the abstract model of agent can be compared to reference models, defined as 'a division of functionality together with data flow between the pieces' [5]. The main difference is that the abstract model is not based on as much experience as full-fledged reference models, although it relies for the major part on well-known resources.

The section first introduces the abstract architecture with a graphical representation, and then links it to the execution model.

4.2.1 Abstract architecture

Figure 4.2 depicts the agent architecture. It is similar to general ones in the agent community, and it introduces specialized elements for exception management. In particular, the elements were introduced so as they can be removed from agents that do not require such functionality or due to design decisions.

4.2.2 Elements of the architecture

The agent architecture contains four main elements to correspond with the execution model, namely the perception, actuation, internal mechanisms, and internal representation.

Perception. The perception element encompasses the *sensory functionalities* and the *evaluation* function. Sensors receive and interpret events from the environment and pass them to the evaluation. This latter element is responsible for estimating the relevance and the appropriateness of the events. An event is categorized as *relevant* by the *relevance filter* when it pertains to the agent, its acquaintances, or its activities, according to the decision of the agent. The event is otherwise discarded as irrelevant, and the agent then returns to the sensor function. Relevance filters are dynamically generated by the *relevance generation* in the actuation element to steer the agent perception strategy (similar to the 'focus' of active perception [110]). Relevant events are further evaluated for appropriateness by the *expectation filter* to distinguish expected events from known and unknown exceptions. Awaited events are defined dynamically by the *expectation generation* function in the actuation element, according to the agent decisions.

Agent internal mechanisms & internal representation. An event and its evaluation (expected, known or unknown exception) are forwarded to the *agent internal mechanisms*. The evaluation uses the *internal representation* element as reference

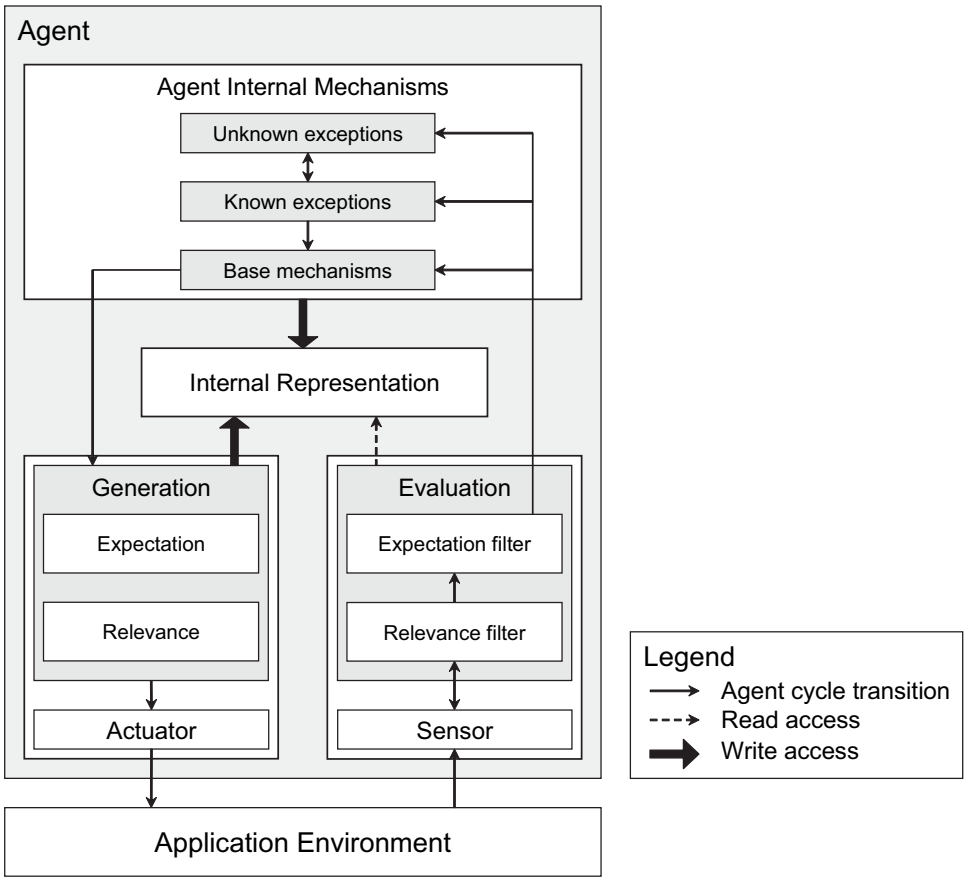


Figure 4.2: Agent base architecture for exception management

to distinguish the events by accessing the agent acquaintance network, for example. The internal representation refers to any representation type inside the agent. For example, the BDI and KGP architectures have a set of knowledge bases [80, 48], whereas some other agents can have simpler internal representations, such as a set of configuration parameters.

The agent internal mechanisms element receives evaluated events and activates one of its three elements, depending on the evaluation. Expected events trigger the *base mechanisms*, whereas exceptions trigger the corresponding mechanisms. The base mechanism can provide facilities such as planners, inference engines, or others such as PRS, MANTA, etc. [19] to deal with expected events. The two exception mechanisms manage the event by setting appropriately the agent internals, so that the base mechanism can handle the case or continue the activities of the agent.

The three mechanisms are interconnected and they just form a reference model of this part of the architecture. Implementations of this model may merge the three mechanisms into a general-purpose one, keep it layered, or simply ignore the exception layers. In particular, existing agent architectures would mostly cover

the functions of the base mechanisms. The abstract architecture only requires that the result of the process ends in the base mechanism and outputs some commands for actions to be taken in the environment (possibly none, e.g. observation mode). The internal mechanisms as a whole exploits the international representation, which contains the agent knowledge including the tabular knowledge introduced in 4.1.2.1, page 53.

Actuation. The action command finally entails the *generation of expectations and relevance* criteria for the next evaluation of percepts from the environment by the Evaluation functions. Typically, changing to the next states of an interaction protocol are added as 'relevant expectations' in the internal representation. In the end, the command is applied in the environment by the *actuator* function.

4.2.3 Correspondence table with the execution model

The following table explicits the correspondence between the elements of the execution model and the architecture. The first column lists the elements of the execution model as presented in the previous section. The second column compiles the elements of the architecture presented in this section. Some of these elements are also the coarse compounds of the architecture, and some of them are grouped in distinguished architectural compounds in the third column.

<table>
<tr><th>Execution Model</th><th>Architectural Element</th><th>Architectural Compound</th></tr>
<tr><td>Message reception</td><td>Sensor</td><td rowspan="3">Perception</td></tr>
<tr><td>Relevance filtering</td><td>Relevance Filter</td></tr>
<tr><td>Expectation matching</td><td>Expectation Filter</td></tr>
<tr><td>Decision process</td><td>Base mechanism</td><td rowspan="6">Agent Internal Mechanisms</td></tr>
<tr><td>Handler selection</td><td rowspan="2">Known exception</td></tr>
<tr><td>Handler preparation</td></tr>
<tr><td>Handler search</td><td rowspan="3">Unknown exception</td></tr>
<tr><td>Handler evaluation</td></tr>
<tr><td>Handler generation</td></tr>
<tr><td>State update</td><td>Generation</td><td rowspan="2">Actuation</td></tr>
<tr><td>Message sending</td><td>Actuator</td></tr>
<tr><td>$\mathcal{K}$ and tabular knowledge</td><td colspan="2">Internal representation</td></tr>
</table>

Table 4.6: Correspondence table between the execution model and the architectural elements, according to Fig. 4.1 and Fig. 4.2

The table allows to show that the abstract architecture covers all requirements for a full implementation of the execution model. The architectural elements and compounds serve two properties in the implementation of an agent with exception management capabilities.

- The compounds provide the designer with a high-level and common architecture model [86]. The columns then guide toward refinements of the architecture and considerations about the exception management mechanisms.
- Elements and compounds allow separating the concerns of application logic and exception logic.

  Application logic. Base mechanism, sensor, actuator, and internal representation ($\mathcal{K}$). One element per compound.

  Exception logic. The remaining elements, with at least one per compound (the internal representation is also part of it in $\mathcal{K}$ and the tabular knowledge).

The two properties are important for the designer as explained throughout the document: The refinement and separation of concerns are recognized good practices in Software engineering.

4.3 Conclusion

The definition of agent exception relies on the notion of 'unexpected event' (c.f. chapter 3). The model of agent execution formalizes this notion in terms of violation of expectations according to interaction protocols. Agents execute interaction protocols in their activities, and they predict what should happen according to the specifications of the protocols by generating dynamically a set of expectations. Inputs that do not comply with these expectations are then deemed as unexpected events, thus triggering exception management mechanisms.

Agent exceptions require specific mechanisms indeed to support designers in writing appropriate management code. Our approach is to elaborate on the agent architecture to develop the analysis of inputs and detect unexpected events, in the case of interaction protocols. The set of mechanisms presented in this chapter becomes part of the agent architecture, so that the tasks of the designer in exception management is concentrated on writing appropriate handlers. The mechanisms exploit handlers to treat exceptions when appropriate.

The architecture endows agents with mechanisms, so that they can deal with exceptions by themselves. Previous work on exception management take similar or complementary approaches that we compare in the next chapter, along with a quantitative evaluation of the computational cost of our approach.

Five

Experiments and Model Validation

The aim of this chapter is to provide a qualitative and quantitative analysis of the agent exception management mechanisms presented in this document. The analysis is based on a series of experiments realized with several implementations of the case study presented in the introduction, from page 11. The purpose of having several implementations is to compare the characteristics of different approaches devoted to the issue of exception management.

The organization of this chapter is as follows. In the first section, the experimental settings are given with a description of the scope of the implementations, the experimental protocol, and technical details. The second section focuses on the qualitative analysis of the work, focusing on comparing the properties of the approach to related work. The third section exposes the numerical results produced by the series of experiments and a quantitative analysis of the model.

5.1 Experimental settings

The experiments rely on several implementations of the case study, each with a specific approach, as an agent-based simulation. In other words, the different implementations provide the same services related to the case study, and they differ on the way to build the system and the mechanisms for exception management.

Exception-free is the reference system, where the agents are ideal and do not cause any agent exception. This system is unrealistic due to the hypothesis of having ideal agents, and it serves essentially to compare the extra computational cost introduced by exception handling mechanisms.

Plain system is the reference system with an ad hoc set of mechanisms to cope with agent exceptions. The approach is ad hoc in the sense that it only relies on usual good practices in software engineering to implement the reactions of agents facing agent exceptions.

Sentinel system is the reference system extended accordingly to the sentinel approach [40]. Although the present sentinel approaches do not verify properties of agent systems, this implementation is introduced to compare such a system-level approach to the agent-level approach proposed in this document.

EMS (Exception Handling System) is an implementation of our approach for the validation purpose.

All the implementations with exception management capabilities are equivalent in the sense that they can cope with the same kind of exceptions during the experiments and they are executed under the same conditions. They differ however fundamentally in the underlying approach.

5.1.1 Scope of the EMS implementation

The EMS implementation covers partially the extent of the execution model. Fig. 5.1 shows the coverage of the implementation over the model introduced in section 4.1.3. The fundamental mechanisms of the agent and the exception management system are included in EMS. The other mechanisms for searching, generating, and evaluating handlers are not included in these experiments, since they are out of the scope of the possible comparisons with other approaches.

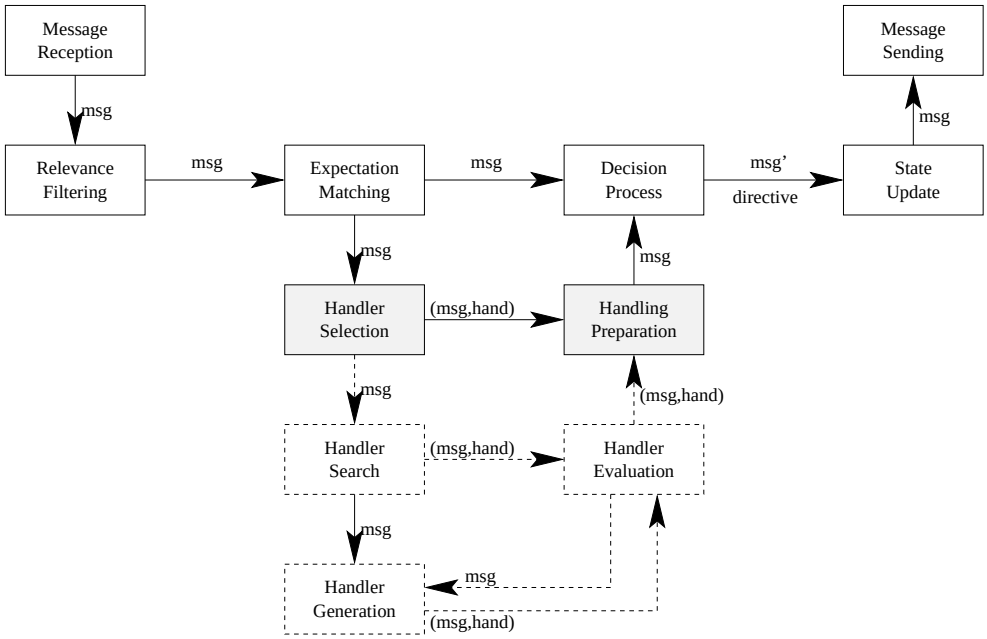


Figure 5.1: Coverage of the implementation (plain lines) over the execution model (plain and dashed lines)

The mechanisms for unknown exception management are in fact part of the EMS, but they do not intervene in the simulations since the situations encountered

by agents are controlled and exceptions are all known, i.e. agents own appropriate handlers for the exceptions that can occur.

Exceptional situation in the experiments. The case study is source of a number of potential agent exceptions. The experiments target a single exceptional situation in the aim to evaluate the overhead cost. The `RefuseDelay` handler leads the agent that uses it to refuse any delay and reorganize the execution of the interrupted protocol. The rationale of this handler is to make the agent focus on its main task and not waste time with delay request from providers. A formal account of this protocol is written with the formula:

$$\begin{aligned}\texttt{RefuseDelay} = (&inform(\text{RefuseDelay}),\\ &\tau(\text{Remove expectation for this agent}),\\ &\tau(\text{Update protocol information},)\\ &end_h)\end{aligned}$$

The handler leads the agent to inform the delay announcer that the delay is not granted. The implementation uses an *inform* performative that returns a refusal based on the request. The agent sends this message and also updates information about the protocol. The expectation related to the announcer is removed from the expectation table as it is not expected to reply on time, and the announcer is removed from the participant list. The handler allows then to return immediately to the interrupted protocol by terminating the handler.

The pattern to select the handler is presented in the following formula, based on the message pattern (c.f. formula 4.1, page 50).

$$(\_, \_, \_, inform, "delay = (\backslash d) + ", \_)$$

Messages that trigger the `RefuseDelay` handler are therefore messages with an *inform* performative and a content that matches the regular expression $delay = (\backslash d)+$. For instance, the expressions $delay = 500$ and $delay = 12000$ are recognized by this expression, whereas $delay = 5.00$ and $delay = Tonight$ are rejected. It is therefore assumed that agents have access to minimal coordination facilities for open systems, including a common ontology to share the concepts of delay and time (in the experiments, time is in milliseconds).

The `RefuseDelay` handler provides the advantage to fit the case study, belongs to the ACK class that makes it non-trivial, and ensures the agent continuity of execution. The `RefuseDelay` handler is however not always the best solution to the agent. When adequate peers for an interaction protocol are rare in the system, an agent *should* accept delays and handle them properly. The remark is important in the sense that the `RefuseDelay` handler is *generic* (it can be applied to other domains), but *not a general solution*. This is the reason why the agent execution model includes a *handler selection* phase that sets forth the preferences of the agent in its context (c.f. algorithm 4.5, page 59). The preference of the agent allows to tune the selection of handler, for example the `RefuseDelay` when the system is crowded, and a looser one when peers are rare.

5.1.2 Experimental protocol

The series of experiments conducted in this research so far followed the process depicted in Fig. 5.2.

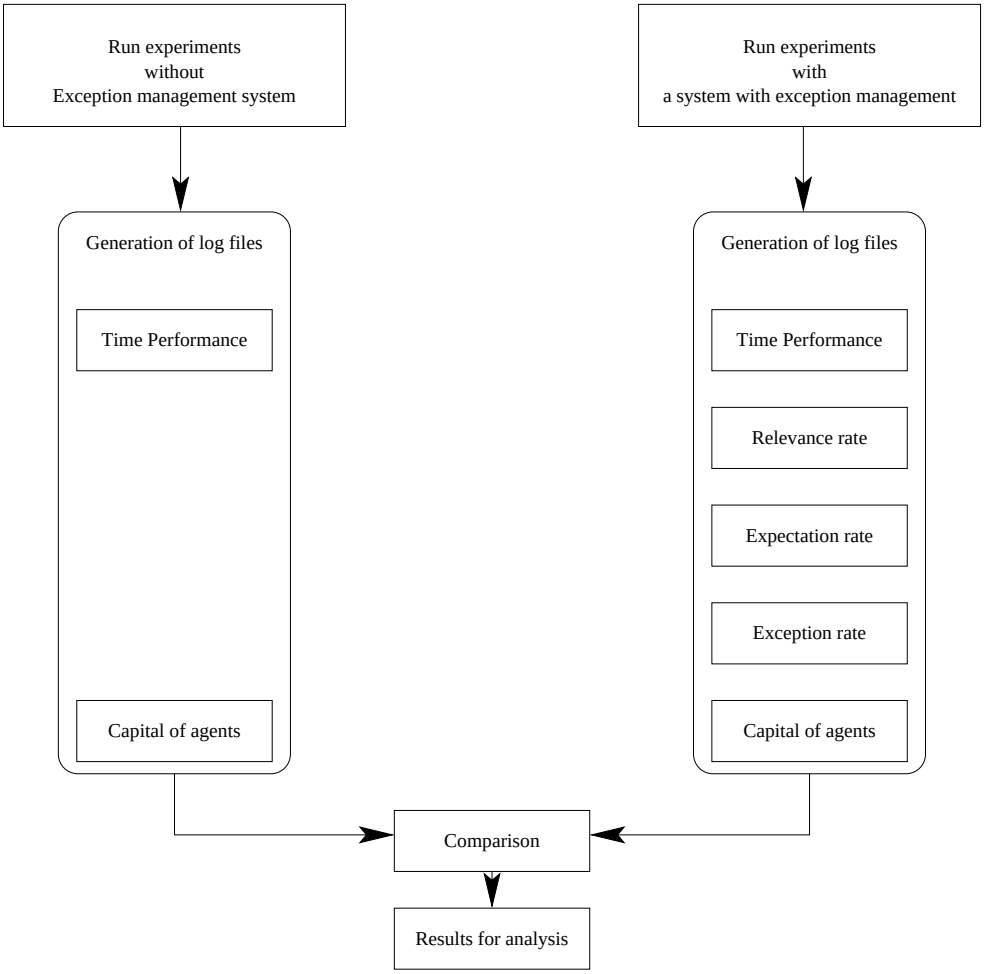


Figure 5.2: Experimental protocol of the experiments

Series of two types were conducted to evaluate the properties of the implementations and the quantitative overhead of the EMS. The first type used the exception-free and plain versions of the system (left-hand side of the figure). Although agents would fail in handling exceptions for the exception-free version, the runs with this system were 'ideal', i.e. the experiments could guarantee the proper execution of the CNet protocol by the agents. The ideal case serves as a reference to evaluate the overhead of other approaches. In practice, the relevance of the ideal case is rather low as it does not reproduce real conditions. The second type of runs used systems with the different exception management approaches (right-hand-side of the figure). The management capabilities were concentrated on

the `RefuseDelay` handler, and this type of exception is artificially generated at runtime with a rate of 5%. In fact, agents announce randomly a delay to simulate an exception.

The two types of experiments were run three times each for one hour to produce statistical results. The initial conditions were the same in all experiments. For example of initial conditions, each agent starts with a capital of 1000 units of currency and with a 'stock' of service to sell in the first place. Production rules for services and requirements are the same in all experiments.

The reason for one-hour runs is explained in the later parts of this section. The basic reason is that stationary states appear after few minutes onward. Experiments were repeated three times only, since very similar characteristics could be observed. All runs produce log files about the different criteria presented in Fig. 5.2. The log files are processed to extract statistical information and compared to evaluate the overhead in practice.

5.1.3 Technical details

The code of the experiments was written in Java 5.0 for portability and efficiency reasons. In order to run experiments on several computers, this programming language was convenient. In particular, the program generates individual log files for agents, and a log analyzer produces additional files with statistical computations. The access to the file system in Java is transparently managed and the program could run on several platforms. An additional reason for choosing this language is that the system has been implemented as a multi-threaded application to ease its debug, maintenance, and modifications for the purpose of the experiments. Java provides good support for multi-threaded applications and ease their development. Finally, the language was chosen among others including Smalltalk and C++. As for the goal of the experiments, none of these languages seems to provide significant advantage over the others, and the ease to use Java and its numerous API settled the choice.

The following code sample in Fig. 5.3 shows how part of the agent architecture has been implemented. The figure shows the complete code of the `handlerSelection` method, which aims at implementing the functionality of the same name in the execution model.

The message that is input to the method is already deemed as an exception. The method aims at returning an appropriate handler, if any. The code creates a tuple based on the message to check against the agent handler table. In the present code, it is required to find only one handler, and other cases are not managed (no preference, no re-direction to the handler search). If no handler is found, the method returns `null`, and it otherwise returns the found handler. The method logs its activity for this agent for analysis purpose.

```
private Handler handlerSelection(Message theMessage) {
 final Handler result;

 List<String> message = theMessage.getAll();

 // Look for handlers
 message.add("nil");

 Tuple mess = new Tuple(message);

 List<Tuple> handlerList = handlerTable.findWeak(mess);

 assert handlerList.size() == 1 : "The selection does not deal with this case yet.";

 if (handlerList.size() == 1) {
  String handlerName = handlerList.get(0).lastElement();
  Handler hand = null;
  try {
   hand = (Handler) Class.forName("exag." + handlerName).newInstance();
  } catch (InstantiationException lException) {
   System.err.println(lException.getMessage());
  } catch (IllegalAccessException lException) {
   System.err.println(lException.getMessage());
  } catch (ClassNotFoundException lException) {
   System.err.println(lException.getMessage());
  }
  result = hand;
 } else {
  result = null;
 }

 logMe("Handler selection done for " + theMessage.toString() + ": "
     + ((result != null) ? result.getName() : "null"));

 return result;
}
```

Figure 5.3: Sample code from the experiments: Agent method `handlerSelection`

5.2 Qualitative Analysis and Comparison

5.2.1 Quality criteria

The selected approaches to exception management are equivalent in terms of functionalities. The engineering properties for the system designers are however different on several issues. The following list compiles the quality criteria that we evaluated in this comparison.

- Separation of concerns
- Maintenance of exception management code
- Robustness to autonomous behaviors
- Tasks for engineering management code

The separation of concerns refers to the distinction between the code of the implementation devoted to the application scenario (to fulfill functional requirements) and the one devoted to exception handling (to fulfill quality requirements).

The maintenance of exception management code evaluates the ease of modification of the code in the application. The robustness to autonomous behaviors refers to the panel of situations that can be managed by the code without change by the designer. Robust code should react consistently, not necessarily appropriately, to a certain range of inputs. On the contrary, code that is not robust is very brittle facing most inputs but the ones explicitly declared by the designer. Finally, the tasks for engineering management code refer to the range of activities that must be completed by the designer to set up appropriate code for exception management.

5.2.2 Comparison

Table 5.1 compiles our comparison of the different approaches to implement the running example. The base system is not presented in this table as it does not implement exception handling facilities.

	Plain	Sentinels	EMS
Separation of concerns	No	Yes	Yes
Maintenance of exception handling code	Low	Med	High
Robustness to autonomous behaviors	Low	Low	Med/ High
Tasks for engineering handling code	Ad hoc	Sentinels and Protocols	Handlers

Table 5.1: Qualitative comparison

The plain implementation does not separate code according to the agent approach. Separation of concerns depends on the implementation language. Agent exceptions can however occur while there is no programming exception. Separation of concerns for agent exceptions then differs from the usual one, and the plain approach, which is ad hoc, does not feature this property. Sentinels and EMS provide specific mechanisms to deal with exceptions, and the separation of concern is verified. Two types of separations are however realized. Sentinels separate application agents from sentinel agents. The former fulfills the functional requirements and the latter implements the quality requirements. The separation is therefore in the agent society, with specialization of agent roles, either application or exception handling agents. EMS separates the code of a single agent into a part for the application scenario (the top part of Fig. 4.1, page 56) and a part for the handling (the bottom part of Fig. 4.1).

Consequent to the separation of concern, the plain implementation is hard to maintain as the code is brittle to changes. The sentinel approach offers better maintainability, since the essential code for exception handling is in the sentinel agents. Sentinel agent must however coordinate with application agents. If the 'interface' code between sentinel and application agents must evolve, the maintainability becomes harder. Finally, the maintenance of the exception handling code in EMS is higher than the other approaches, due to the execution model that provides an application-independent mechanism to exploit handlers at runtime automatically. Maintenance can thus target the mechanism or—independently—handlers that are attributed to agents.

The robustness of the different approaches to autonomous behaviors is one of the main motivations in this work. Plain and sentinel approaches are recognized as brittle when facing non-collaborative behaviors in agents (e.g. refuse to participate in handling a case). EMS provides agents with individual mechanisms, so that they acquire some robustness facing other agents. Depending on the number of handlers available and the default handler, agents can at least terminate flawed protocol executions, remain in a consistent state, and continue their executions.

The tasks of the designer to engineer the exception handling code differ depending on the approach. The plain implementation is ad hoc, i.e. different designers may use different methods, with various qualities and drawbacks that are difficult to evaluate and track. The sentinel approach reduces the tasks to the design of appropriate sentinel agents and coordination protocols between application and sentinel agents. The guidance provided by the sentinel approach does not let the designer with an ad hoc method anymore. EMS reduces further the tasks to the creation of handlers only. Handlers design is similar to protocol design in the frame of this work, so that we can expect the task to be easier than for the sentinel approach.

5.3 Experimental Results

The experimental results pertain to the exception-free, plain, and EMS versions of the system, so as to compare the different approaches quantitatively. The results are presented as graphs and statistical information obtained from the logs.

The first part of this section presents the results for the exception-free and EMS versions in detail. It evaluates the overhead cost of introducing exception management mechanisms in agents. The second part of the section aims at comparing the plain and EMS versions, focusing on the numerical values. The comparison shows the overhead cost of the EMS version over the plain one.

5.3.1 Overhead cost of exception management mechanism: Exception-free and EMS versions of the system

Runs with the exception-free system produced logs that led to Fig. 5.4. The figure shows a typical time-series of the performance of an agent in the market. The

performance is represented as the number of execution cycles completed by the agent per millisecond.

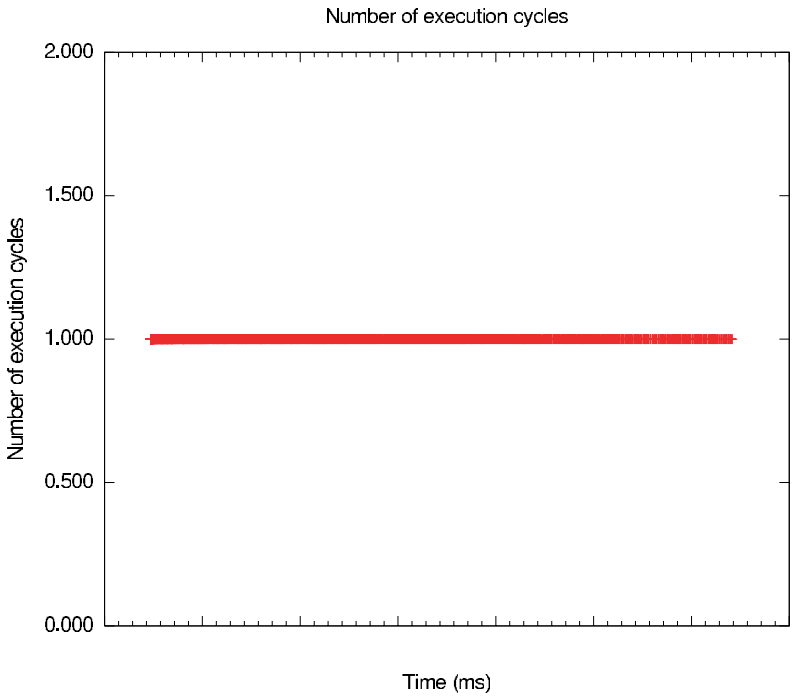


Figure 5.4: Number of execution cycles completed by agent 'Machine Assembler 1' - No EMS

The figure shows that the agent executes once per millisecond with a constant frequency. At the beginning of the curve, the agent is in fact executed more than once per cycle: The log file reveals that the agent could execute twice in one cycle at two occasions in the few first cycles of execution (this information does not appear on the figure plot). Otherwise, the agent executes only once, which is ensured by the scheduler of the multi-threaded application since all agents are threads of the same priority (the Java virtual machine used in these experiments did not guarantee fairness though). One reason for multiple cycles at the beginning is the competition for the processor time. The agents compete less at the beginning for the time their threads and activities all start. The individual curve of each agent was drawn to observe all agents have similar curve profiles, which confirms the explanation of the processor time competition.

Similar profiles are also observed for agents in the EMS experiments, with similar explanation. Fig. 5.5 shows however that the frequency of executions decreases over time to reach a stable value after about two third of the time. It can also be observed that the execution frequency is lower with the EMS, which means the agent executes less often. This observation results from the overhead cost of the EMS that is evaluated in this section.

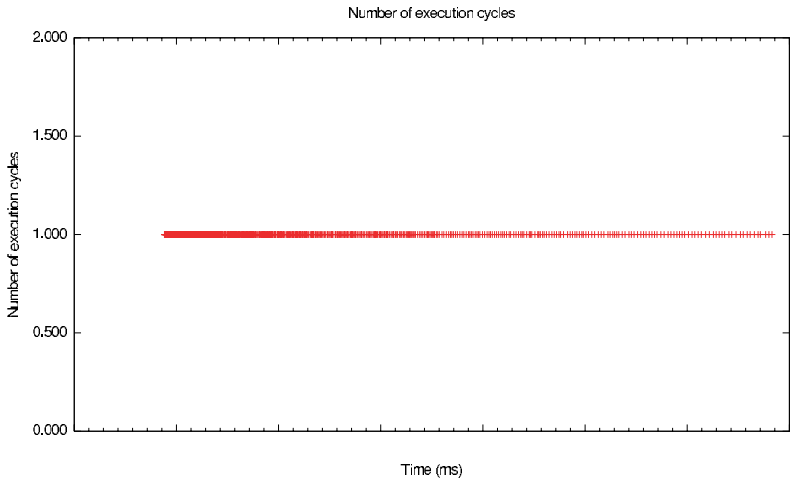


Figure 5.5: Number of execution cycles completed by agent 'Machine Assembler 1' (red) - With EMS

Another run with the EMS had an interesting irregularity: In the plateau of activities (right part, about the last third of the time), two agent threads stopped prematurely due to memory shortage (problem of configuration of the simulation parameters). Other agents continued their activities until the end of the simulations. The irregularities causes an increase of the frequency in the execution of agents for few seconds, and then a return to the almost constant frequency before the irregularity. The occurrence of such event allows to state that agents execute less because of a constant rate of activity. In other words, agents do not lack computing resources but execute their activities according to their constraints (e.g. enough money or machine parts for the production). This leads the analysis to distinguish between the first half of the execution time, where competition for computing resources is high, with the second half, where computing resources are more available.

The following two figures present an averaging that allows to better approximate the cost of each approach. Fig. 5.6 and Fig. 5.7 represent the average execution time of agents in the system over a period of 100ms. In other word, each 100ms plateau is the average number of agents that execute at the same time. Although the two profiles are similar, several runs show the agents in the exception-free system execute more on average. Numerical values and rates are compiled in table 5.2 for quantitative analysis.

The maximal values presented in the table show the difference between the experiment types. 56.8% is the maximal performance with EMS against the reference, which means the EMS divides the maximal performance by a factor 1.7.

The minimum value is almost unchanged in both experiments, which confirms the observation with individual curves: The processor and the Java virtual machine ensure that agents eventually obtain processing time. The overhead of the EMS

Type	Max Average	Min Average	In Stationary Interval		
			Max delta	Max	Min
Exception-free	3.57	1.0	0.09	1.09	1.0
EMS	2.13	1.0	0.04	1.04	1.0
Ratio ($\pm 10^{-2}$)	0.568	1.0	2.25	1.05	1.0
Inverse ($\pm 10^{-2}$)	1.761	1.0	0.45	0.95	1.0

Table 5.2: Comparison of the performance characteristics

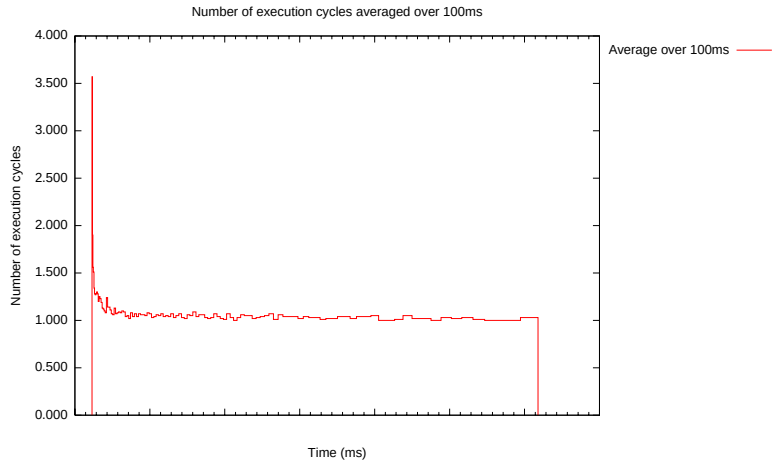


Figure 5.6: Average number of execution cycles completed by agents over 100ms periods - Exception-free

is therefore bounded, since the corresponding agents are run at lest once in each period.

As for the stationary plateau, the values in the table are taken from half-time onward. After the high activity at the beginning of the market execution, the system reaches a stable state, which depends on the initial conditions (capital and stock of agents). In the result table, the maximal values are of the two systems are close (5% difference according to the ratio). One interesting value is the difference between the maximal and minimal values in the plateau (delta). Despite the apparent reduction of the difference between the two types of systems in the long run (on average, agents execute a similar number of times), the EMS has still a significant cost since the delta value differ by 55%.

The following reports show a detailed analysis of the average performance of agent over one execution cycle only. This information complements the one presented so far and gives an accurate estimation of the raw computational cost. The 'Mean average' is the average of the mean execution time of each agent, and the 'Mean deviation' is the standard deviation around the average value. Results

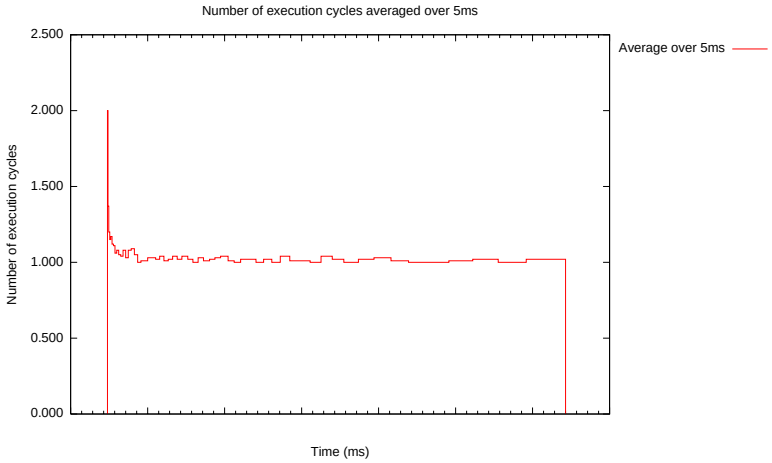


Figure 5.7: Average number of execution cycle completed by agents over 100ms periods - EMS

are given in milliseconds. In the exception-free version, the following results are obtained from the logs.

```
Mean average: 2345.1369627596932
Mean deviation: 530.0239370593646
```

The average execution time is therefore centered around the rounded value 2345ms and its deviation is around 530ms.

In the EMS version of the execution, the results become as follows.

```
Mean average: 5080.905346085171
Mean deviation: 1535.1855354055572
```

The average execution time becomes close to 5081ms and the average deviation is around 1535ms. The raw cost is therefore about 2.17 more expensive with the EMS on average. The standard deviations with and without EMS are similar (between 22% to 30% of the mean values), so the rate of 2.17 can be considered as meaningful indicator. The extra cost of 2.17 is significantly expensive, but it can seemingly be reduced by optimizing the data structures used for the knowledge of agents. In the present experiments, agents process their knowledge data structures as tables (straightforward implementation of the knowledge structure of the model on the software architecture), and most steps in the EMS require expensive look-up through them.

The values obtained can finally serve to compare the theoretical computational cost to numerical analysis. The complexity analysis in table 4.5 (page 62) leads to the following estimation, with N_{DP} the complexity of the reference system, $O(n_k)$ the complexity for Handler Selection, and $O(n_{pro})$ for Handler Preparation. The complexities are related to the execution time per cycle, so that the estimation is

based on the order of execution time. Other measures showed that the average size of the agent knowledge tables was of constant in the experiments (agents execute on average once per cycle).

Execution type	Theoretical Cost	Order (ms)
Exception-free	N_{DP}	10^3 (2345)
EMS	$N_{base} = max(N_{DP}, O(n_{pro}))$	$max(10^3, O(1))$
Known exception	$N = max(O(n_k), O(n_{pro}))$	$max(O(1), O(1))$
Total estimation	$N_{base} + N$	10^3
Measured Value		10^3 (5081)

Table 5.3: Evaluation of the theoretical complexity

The theoretical value has the same order than the measured one. The original analysis predicted that the introduction of the EMS only would increase the complexity by one order, which is not that costly in practice since the experiments are based on the software architecture instead of the execution model, and the activity of agents were restricted to run only few protocols simultaneously.

The study of the time performance of agents show the influence of the EMS on the agent execution cycle. In addition to the absolute performance of the computational cost, the next section exposes other aspects of the performance of agents, namely their capital and their knowledge about the running activities.

5.3.1.1 Comparison of the capital of agents

The comparison between the two types of experiments is also based on the 'money' (unit of currency) exchanged in the market during the execution, which gives a 'social' dimension of the performance of agents, in addition to the raw performance in time and processor.

Agent exceptions impact primarily the activity of the market, i.e. service and money exchanges. The way money evolves over time is therefore expected to show some variations when exceptional situations occur.

Fig. 5.8 and Fig. 5.9 first show the evolution of the money for one agent in the market. The red curves represent in each case the raw data from the logs, and the green curves represent a Bezier approximation of the raw data. The type of approximation was selected as it adequately reproduces the tendencies of the raw data over time in this case. For this reason, the next curves will mostly use the Bezier approximation, unless raw data shows interesting features.

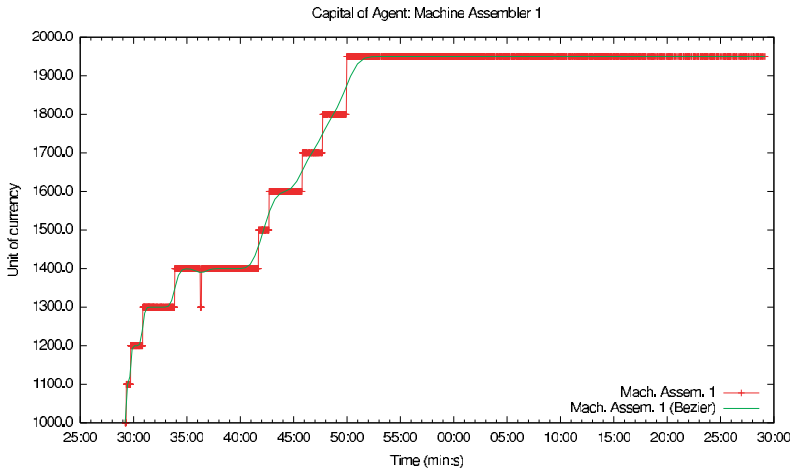


Figure 5.8: Capital of agent 'Machine Assembler 1' over time (red), and Bezier approximation (green) - No EMS

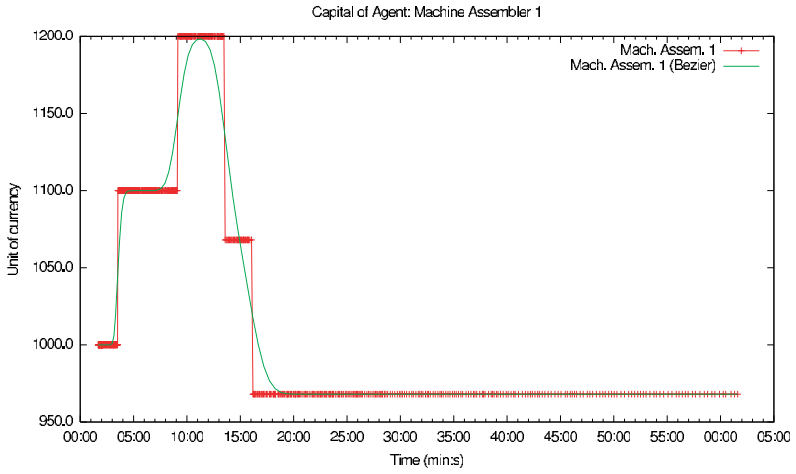


Figure 5.9: Capital of agent 'Machine Assembler 1' over time (red), and Bezier approximation (green) - With EMS

The curves of the agent in the two situations are similar on these runs. In the context of the whole system, Fig. 5.10 and Fig. 5.11 superimpose the curves for each agent and for the average in the two types of experiments. The average is represented both in raw and approximate forms to show the tendency of the whole system over time. The two present curves of individual agents 'Machine Assembler 1' appear also on these curves to show their evolutions in the context of the system.

The two types of experiments produce different system reactions over time. This observation was verified over repeated runs of the experiments with same initial

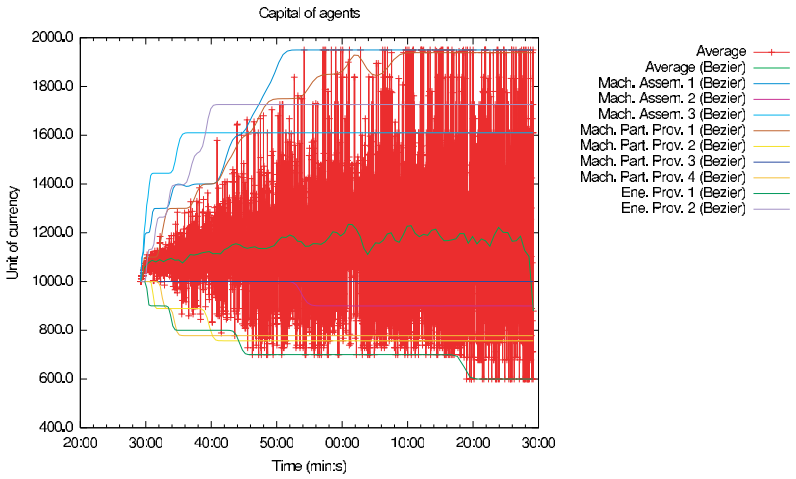


Figure 5.10: Average of the capitals of agents over time (red), and Bezier approximation for the average and each agent (other colors) - No EMS

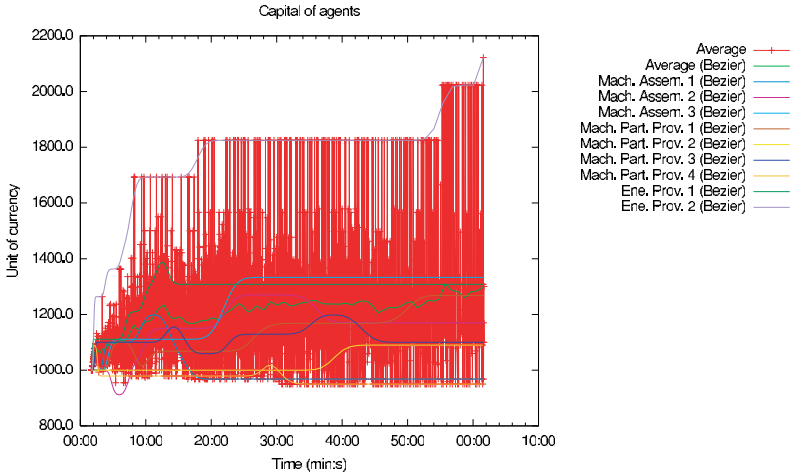


Figure 5.11: Average of the capitals of agents over time (red), and Bezier approximation for the average and each agent (other colors) - With EMS

conditions: 3 runs were used for the final results presented here, but similar system reactions were observed each time.

The interesting common points in all runs is the number of change in the monotonicity of the curves. In the exception-free version, Fig. 5.10 shows that the monotonicity changed at most twice, and for only one agent. In the EMS version, Fig. 5.11 shows curves change monotonicity several times and at any time during the experiments. Although changes occur more often at the beginning, they are still observable at the time of the aforementioned stationary plateau. Runs of several experiments for each type confirm the tendency to have more monotonicity changes

with EMS than without it.

Changes in monotonicity are then a possible consequence of introducing exceptions in the system. The dynamics of agents are modified by exceptions and the EMS, so that their reactions and outcomes in the market differ from the exception-free runs. The relation of the effect to the EMS leads to compare the occurrence of exceptions in the system with the profile of the agent capitals. The next Fig. 5.12 and Fig. 5.13 show the occurrences of exceptions over time and the superimposition with the agent capitals respectively.

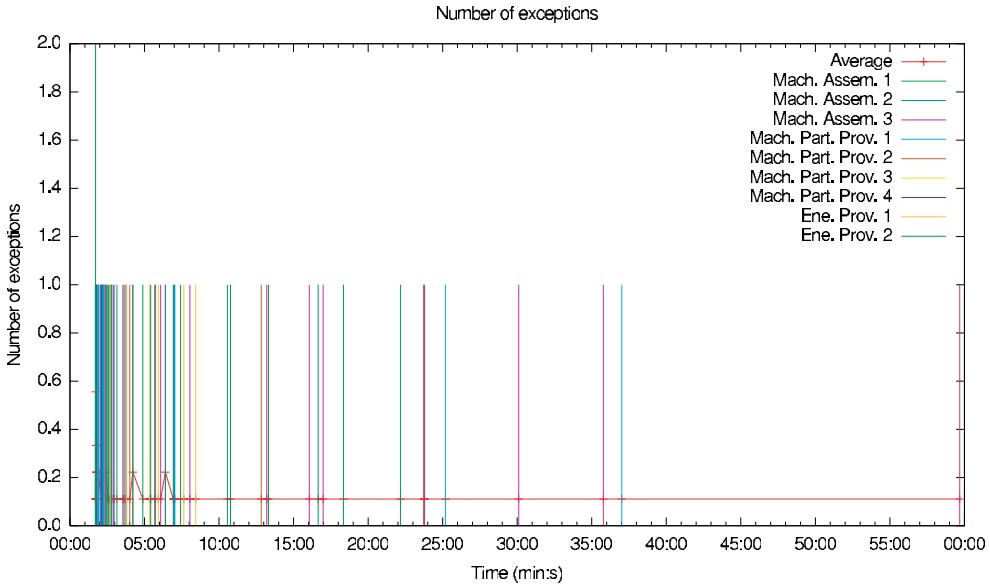


Figure 5.12: Average number of exceptional situations in the agent activities over time (red), and number of exceptions recognized by each agent (other colors) – With EMS

The observation of the exception spikes over the capital curves show a possible correlation between the occurrence of exceptions and the change in monotonicity. All curves are not impacted by exceptions at each spike. Unfortunately, the spikes influence agents that are involved in unrelated protocols, some without exception occurrence. The log files confirm with figures that the possible correlation cannot be verified.

```
22:15:31: New turn, Money=1000
22:15:31: Rel. ok (cnet19, mp3, ma2, inform, [delay=500], 97)
22:15:31: Exp. NOK (cnet19, ...)
22:15:31: Exception detected
22:15:31: Han. sel. (cnet19, ...) done: RefuseDelayHandler
22:15:31: Han. prep. (cnet19, ...)
22:15:31: Proc. (cnet19, ...)
22:15:31: Start handler...
```

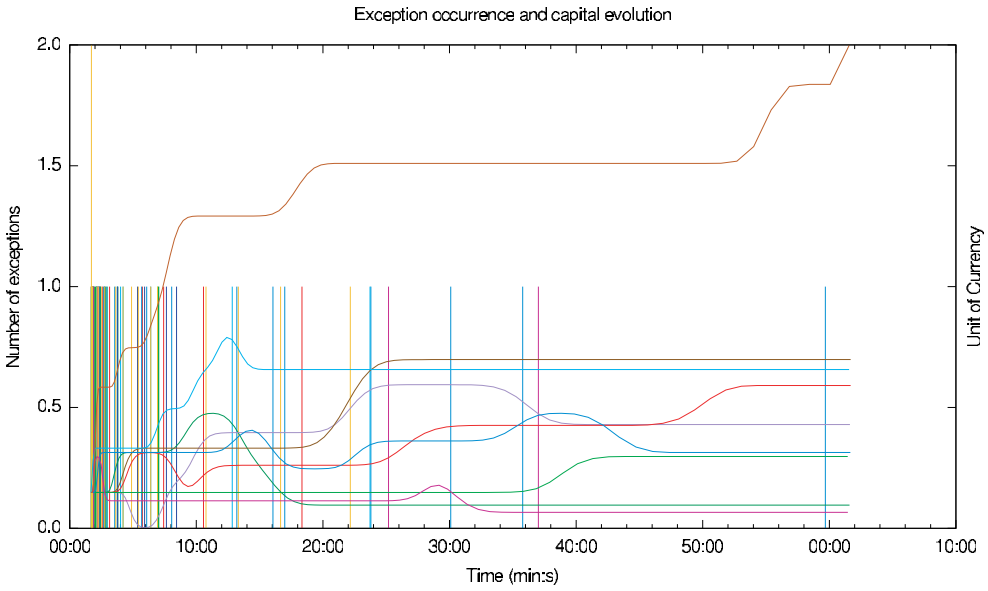


Figure 5.13: Evolution of the capital of agents and exception occurrences

```
22:15:31: Number of directives for this turn is 9
22:15:31: Protocol ended by RefuseDelayHandler44910899
22:15:31: Remove expectation: [cnet19,mp3,ma2,nil,nil,nil]
22:15:31: Upd. (cnet19, ma2, mp3, inform, [nope], 284)
22:15:31: Msg sent: (cnet19, ma2, mp3, inform, [nope], 284)
...
22:15:31: New turn, Money=1000
```

The extract from the log file of one agent shows no influence on the capital over one turn despite the occurrence of an exception.

In conclusion of the study of the capital of agents, the EMS has apparently no influence on the evolution of the capital over time. The EMS has however a concrete influence on the continuity of the execution of the agent, since it allows the agent to continue its execution despite the occurrence of exceptions. Without EMS, the same agents terminate or enter 'infinite loops' expecting messages that can never arrive. The infinite loops can be interrupted by the addition of timeouts, but whenever such timeout is omitted, agents without an exception management mechanism do not behave adequately. The EMS allows agents to continue their activity, thus contributing to the future evolutions of their capitals.

5.3.1.2 Other mechanisms of the exception management

The experiments results allowed to collect detailed information at each step of the EMS phases. The graph of exceptions presented in the previous section showed

the information related to the Handler Selection and Handler Preparation phases, which are necessarily used together in the settings of the case study.

In this section, results about the Relevance and Expectation filtering phases are presented and related to the performance of agents. Fig. 5.14 and Fig. 5.15 show the graphs of the two phases over time.

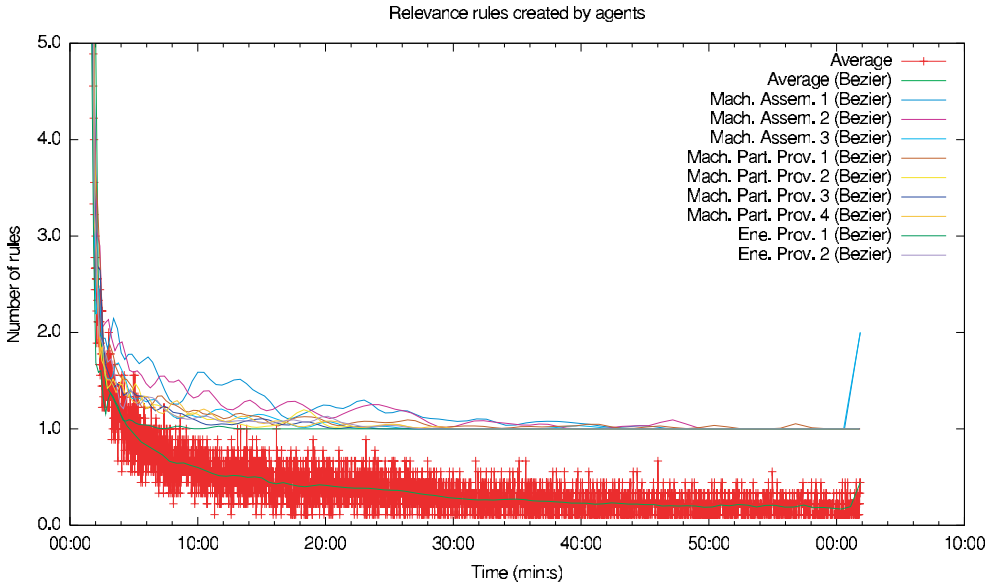


Figure 5.14: Average number of relevance rules generated by agents over time (red), and Bezier approximation for the average and each agent (other colors) - With EMS

The profiles of relevance and expectation graphs are very close since both are produced and remove at the same time in situations like the creation or termination of a protocol. The graphs are however not identical since expectation rules can be created during the execution of a protocol, while the relevance remains the same all along the protocol (see Alg. 4.3, page 58). For example, a client creates only an expectation for the `result` performative when it sends an `acceptProposal` message to the provider that won in the protocol. No relevance rule is created at that time.

The graphs show the period of high activity in the system. When rules are created in higher number, it means more protocols are run simultaneously, whereas fewer rules indicate a slow down in the agent activities. In the plateau of activities introduced in the performance evaluation (Fig. 5.7), the production of rules is one per cycle in accordance with the single execution of the agent on average.

The graphs allow to illustrate the reason for the overhead cost of the EMS over exception-free version of the system. The maintenance of relevance of expectation rules that serve in the EMS mechanism demand a significant computation. As a result, the consequent cost can be controlled by optimizing the management of the

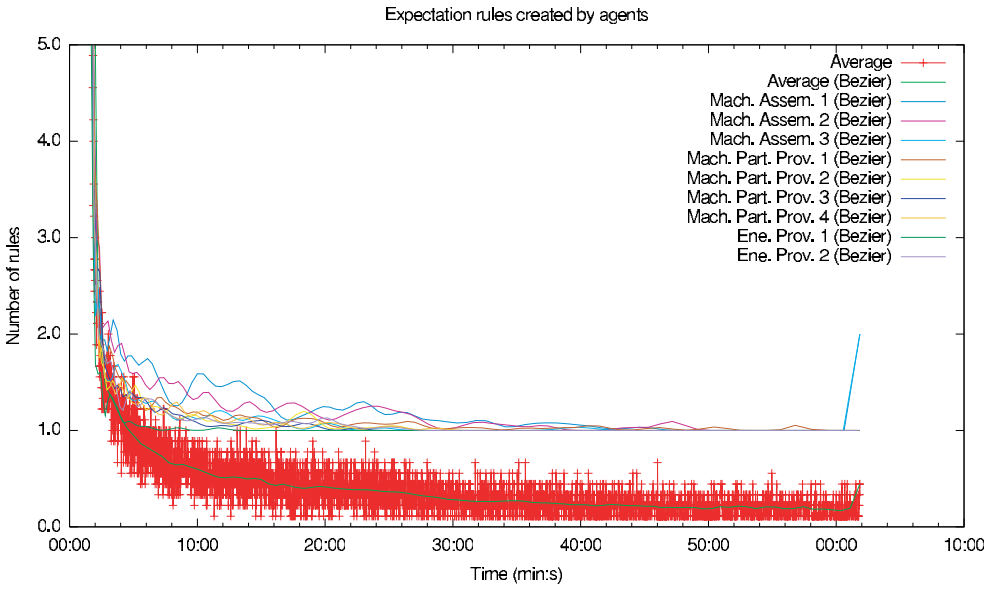


Figure 5.15: Average number of expectations rules generated by agents over time (red), and Bezier approximation for the average and each agent (other colors) - With EMS

rules with appropriate algorithms, although this cost cannot be eliminated.

5.3.2 Comparison between the Plain and EMS exception management systems

The aim of this comparison is to present the different costs of the Plain and EMS versions of the system. A difference is expected owing to the type of approach implemented. The Plain version is especially tailored for the case study, and the exception management is integrated based on standard practices in Software engineering. In other words, the Plain approach is 'optimized' for the case study. On the other hand, the EMS version is based on our general mechanism to let agents deal with agent exceptions. The EMS-based system is therefore 'less optimized' and we expected a higher overhead cost. The present part details the comparison with the results extracted from the logs of the experiments.

Tables 5.4 and 5.5 provide performance results of our implementations in the case of the plain and EMS versions of the system. The exception-free performance is shown again as a reference.

The plain approach is close to 95% of the performance of the exception-free systems, which confirms that this implementation is close to optimal in terms of the overhead cost of an exception management mechanism. On the other hand, the performance of the EMS version is around 63% of the Plain one. The EMS approach is therefore 47% more expensive than the Plain one.

Approach	Max	Min	Max in Stationary Interval
Exception-free	3.57	1.0	1.09
Plain	3.37	1.0	1.08
EMS	2.13	1.0	1.04

Table 5.4: Comparison of the performance characteristics

Approach	Time ($\pm 10^{-2}$s)
Exception-free	2.35
Plain	2.35
EMS	5.08

Table 5.5: Average computational cost of an agent cycle in terms of execution time

The plain implementation has no significantly different cost with the exception-free system. The EMS version costs consequently 2.17 more time to complete an agent execution cycle (including deliberation), which is the same numerical value as the previous section. The explanation is the overhead cost introduced by management of relevance and expectation rules, which does not exist in the Plain approach. Since the Plain approach allows to perform exception management similarly to the EMS at runtime, this result confirms that the management of the rules is the key factor to optimize the EMS and still leverage the generality of the approach over the Plain version.

5.4 Conclusion

The analysis presented in this chapter compares different approaches that deal with exception management in MAS. The purpose of the EMS approach is to endow the agents individually with appropriate capabilities with regards to exceptions, and to comply with the characteristics of agents, notably their autonomy. Other approaches verify some of the agent characteristics, but the comparison shows that only the EMS deals adequately with the autonomy matter (by design of the approach). Consequences on engineering agents with exception management capabilities are also examined. The main advantage of the EMS over other approaches is to be more robust and to reduce the tasks of the designer, thus reducing the load and focusing on essential handling mechanisms. On the other hand, the contribution of the EMS to the design has a cost at runtime, so that designers have to evaluate whether the target system can cope with the additional cost. For instance, the design of systems on resource-limited platforms (e.g. mobile devices) might prevent from using the EMS. The overhead cost is however bounded, as shown in the experiments, and an optimized management of relevance and expectation rules can allow to reduce this cost, such that specialized versions of the EMS might be

adapted to resource-limited platforms.

Six

Conclusions

Multi-agent systems are expected to feature many qualities in terms of flexibility, robustness, and perhaps more generally, the capability to adapt automatically to the dynamics of its agents and its environment. Exception management is among the mechanisms that participate in the realization of these qualities, and the agent research community has produced models and techniques to endow MAS with exception management capabilities.

The past research has focused in the first place on the systemic dimension of exception management. The notable achievements of Hägg with the sentinel agents and Klein et al. with the reliability database are significant contributions to exception management at the level of the system: Both approaches rely on introducing exception-oriented services in the environment of MAS [40, 53].

The other approach developed in this document is at the level of agents: How can designers introduce the qualities of flexibility, robustness, or adaptability in MAS in case the sentinels or the reliability database fails? In other words, the motivation of this second approach is to endow individual agents with exception management capabilities. The capabilities allow the agent to continue its activity and to remain in a consistent state despite the occurrence of exception, independently from external services.

The two approaches are complementary in their benefits to MAS. The original work at the system level deals with coordinated exceptions efficiently, owing to a central or decentralized service that 'orchestrates' the management. The work at the agent level allows to deal with individual and coordinated exceptions in a distributed fashion, which is more complex—therefore less efficient [101]— but also more robust and flexible when parts of the system encounter exceptions. The system level contributes to the efficiency and the agent level work *palliates* the issues of robustness at the system level, primarily due to the agent *autonomy*, and the system *openness* and *heterogeneity*.

6.1 General contributions of the present work

The general contributions of the present work are first to define the notion of *agent exception* in the context of Multi-agent systems. Past research has succeeded in setting forth the intuition of agent exception, but no work had proposed so far any definition of the nature of an exception in agent systems. The definition of this document is elaborated in the context of agents that execute protocols, but the fundamental notion in the definition is the *unexpected character* of an event. Other types of agents can probably reuse the present work provided a proper interpretation of the term 'unexpected' is chosen. For example, planning agents could seemingly leverage the approach developed in this document if 'unexpected event' is understood as an event that is not indicated in the plan. This example is very close to the protocol approach, but it shows an instance of transposition of this work to another type of agent.

The *preservation of the agent autonomy* is the second general contribution of this work. Past research on system-level approaches recognize a common limitation: Agents must collaborate during the exception management procedure [53]. The assumption of agent collaboration is however strong within a society of autonomous agents. The main consequence of autonomy is the inability to predict a collaborative situation. Besides, collaboration is a reasonable assumption in the rationale for creating a system [41, Chapter 7], but it hides numerous issues for exception management in MAS. Agent can be collaborative but fail for unexpected reason, thus having a 'non-collaborative' behavior in the context of an activity [6]. The preservation of autonomy as a condition provides foundations to deal with the non-collaborative case. The work presented in this document is under a strong condition of agent autonomy so that the models and facilities elaborated in this research allow agents to continue their activity and remain in a consistent state despite non-collaborative situations. Agents are able to decide the termination of an activity and continue others independently from the decision of other agents. The actual handling of such situation depends on the handlers available to the agent, and the foundation guarantees that such handlers are found and used by the agent with respect to its knowledge and autonomy.

The last general contribution of this work is the preparation of the agent execution model and corresponding architecture for integration with some other works akin to exception management. The proposed model deals with agent exceptions at the level of agent, while related work focused on the system. The model features two 'connection points' where both approaches can complement. First, the execution model is built on a classical agent cycle with perception, process, and actuation. The same assumption is done in other research so that the execution model can accept messages from external entities that support exception management. Such messages can inform the agent about system events, such as network congestions, or more intricate situations, such as a circular wait deadlock. The external entity is then an event notification service, as can be observed in the agent community [115, 78, 79]. Other messages can also advice the agent to act in a particular

way, for example to break a circular wait deadlock. The second connection point is the *Handler search* phase of the agent. When the agent reaches this phase, it attempts to poll agents and other alternative external entities to obtain advices on how to handle a situation [51, 82]. Such situation does not infringe the autonomy assumption in the sense the agent remains independent on the application of the received advice. In this phase, the usage of external entities is explicitly considered as an appropriate action to take by the agent, thus exploiting the system-level facilities.

6.2 Contributions to Agent-Oriented Software Engineering

Software engineering is one of the domains of application of the present work. Exception management systems in software exist since early programming languages, and they evolved with the increasing complexity and the novel challenges of modern software systems. The apparition of exception models in distributed computing illustrates such evolutions.

Multi-agent systems constitute a comprehensive view on modern systems that embrace the complexity of large-scale distribution over nondeterministic environments such as the Internet or wireless networks. The dependability of MAS is essential for the development of agent technologies, and exception management should therefore deal with them. As software, MAS can leverage the past achievements in distributed systems, although the complexity, openness, and heterogeneity require further research. MAS need however specific support as systems of autonomous agents, and the present work is thought of as a contribution to this endeavor. Concrete contributions to Software engineering is to consider the abstract notion of autonomy as a guidance to create exception management systems. An essential concept in the exception models of programming languages and distributed systems is the *context* of an exception (or syntactic unit, historically). In the example of Java, such context is determined by the peer keywords `try/catch`. In the following code sample, the context of the exception is determined by the curly brackets between `try` and `catch`, i.e. the syntactic units introduced in chapter 2 [34].

```
try{
    //Do something
} catch (Exception theException) {
    //Handle the exception
}
```

In distributed computing, the context is a joint activity between process, such as the Coordinated Atomic Actions [112]. Agent systems naturally lead to consider the agent itself as context for an exception due to the notion of autonomy. The autonomy then becomes a criteria of modularity for systems that can be useful for analyzing system architectures. The modularity of MAS is a consequence of autonomy and it

appears as a possible assumption that should be chosen to engineer systems that have to interact with other systems, built by unknown designers.

Finally the work presented in this document follows a tradition in Software engineering to *separate the concerns of application logics and exception logics* embedded in programs. Past research in MAS already set up such separation, but the contribution was then at the system level. Agents were considered as the application logics and external entities provided the exception logics. In the present work, the separation of concerns is established at the agent level, and it complements the system level separation. The two levels of separation are then another view on the modularity of MAS as for exception management.

6.3 Contributions to Distributed Artificial Intelligence

The contribution to Distributed Artificial Intelligence (DAI) is threefold. The agent execution model is first an attempt in Artificial Intelligence to create a model that explicitly separates the general-purpose reasoning capabilities of the agent from mechanisms devoted to exception management. The separation is important owing to the numerous agent models that already exist: The exception management system is a separate extension that can be 'plugged' to a general-purpose reasoning model. The 'plug-point' is then the Decision process phase in the agent execution model. Some models such as the KGP model of agency are already capable of adapting to exceptional situations [48, 98], and the major contribution of the present work is to set forth the autonomy and a model that explicitly separates the logics, as aforementioned in the Software engineering section.

The second contribution of this work is to position work in DAI on the topic with work in AI. The properties of MAS led to distinguish system- and agent-level exception management. The former pertains primarily to DAI, and the latter to AI. The complementarity of the approaches was discussed throughout this document, which shows that benefits can be expected from a future synergy of techniques from the two points of view.

Finally the third contribution is the technical framework provided by the agent execution model and the corresponding software architecture. The software architecture serves in the first place to guide the implementation of the execution model, which allowed straightforward applications such as the Energy market case study. The execution model is furthermore a framework in AI, as several mechanisms of AI such as Abductive reasoning are relevant to develop some of the phases of the model, notably the Handler generation.

6.4 Future perspectives

The agent execution model presented in this document settles a number of future perspectives. The model is a framework and the current coverage of the study does

not address all the underlying research issues. This last section aims at presenting the work that is either on-going or extending the current state of study.

Integration of approaches and work on handlers. The two main on-going activities are the practical integration of the work with models that rely on external entities, and the generation of handlers with abductive reasoning. Although the two levels are complementary, the pragmatics of this claim are not studied in detail yet. One of the underlying research issues lies in the *evaluation* by the agent of a handler received from an external entity. The Handler evaluation phase is then the key phase, and the present achievements in this work are limited to a simple case: The evaluation just checks that the external handler leads to a desired state eventually. This case can apply to simple cases, but it becomes hazardous in real settings, i.e. in open and heterogeneous settings. One possible target achievement is for the agent to check that each step of the handler is acceptable in a computationally economical way. Related work about this topic is for instance the analysis of generic protocols[1] by the agent to adapt it to its specific concerns.

The generation of handlers is a challenging issue that has the potential to make agents more robust in unknown situations. The purpose of the generation should not be the production of any kind of handling, but it should at least focus on mechanisms that maintain the agent continuity, i.e. maintaining its activities in a consistent state. This minimal requirement is essential in the agent execution model and current work aims at exploring different ways to generate useful handlers in an economical way. This minimal requirement is also the reason why a handler that 'ignores' a message seems appropriate as default. Ignoring a message cause no harm to the agent state and can maintain activities. The ignore handler is however insufficient whenever the exception has a real impact on the activity. A supplementary analysis is required to ensure that ignoring should have no side-effect.

Two ways are currently considered for handler generation, namely Case-based (CBR) and Abductive (AR) reasoning models. CBR allows more flexibility in the agent and appears as an economical way to generate handlers from an existing handling knowledge base. In practice, CBR allows to generate handlers slightly different from existing ones, but adapted to a new situation. For example, two handlers can have similarities and CBR techniques are possible approaches to deduce one from the other and some additional knowledge. The problem of a CBR-approach in the context of the execution model is that it is usually difficult to guarantee a sound handler for the social context of agents. The preliminary investigation on the topic cannot permit to conclude at present. As for AR, the approach appears more flexible and sound. One possible way to exploit an abductive reasoning framework for handler generation is to generate 'abducible hypothesis' from the unexpected event and the impacted protocol. The hypothesis allow the agent to 'simulate'

[1] A handler is thought of as a generalization of a protocol in this work, due to the inclusion of 'internal actions'.

internally the possible evolutions of the activity in the next stages. If the series of simulated actions leads to a desired state and complies with constraints of the agent (e.g. time), then the series becomes a handler, and the hypothesis can be assumed. Past work on agent models show the potential of abduction [88, 87, 48], but the technical issues of such approach are numerous and the present research remains in a premature stage.

Further issues. The concept of *nested exceptions* has not been explicitly presented in this document. Nested exceptions are encountered during the handling of another exception, thus requiring the suspension of a handler and the start of another one. The agent execution model *implicitly* supports this procedure. The implicit support comes from the close representation and properties of protocols and handlers in the framework. When a handler is executed, it produces some expectations that must be verified otherwise causing another exception, similarly to protocols. Handlers can then be suspended and resumed as protocols in the handling of nested exceptions. The present work does not however study the case of nested exceptions in detail, due to the strong similarity of their managements.

Further research shall eventually be conducted beyond the framework settled by this document for agents that execute according to protocols. The present work explicitly focuses on these agents owing to the target applications to agent-oriented software engineering. Other models do exist, such as interactions based on dialog or argumentation, and they require appropriate adaptation of the present mechanisms. This document has however identified a key issue for exception management in such models, which can serve as a starting point and a way to reuse the present work: Another model should determine adequately what is an *unexpected event*.

Bibliography

[1] Agent Unified Modeling Language (AUML) Web-Site. `http://www.auml.org/`. 12

[2] FIPA Modeling: Interaction Diagrams. http://www.auml.org/auml/documents/ID-03-07-02.pdf. 50

[3] Autonomic Computing. `http://www.research.ibm.com/autonomic/`, Accessed in October 2006. 3

[4] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental Concepts of Dependability. In *Third Information Survivability Workshop*, pages 7–12. IEEE, 2000. 2

[5] Len Bass, Paul Clements, and Rich Kazman, editors. *Software Architecture in Practice, Second Edition*. Addison-Wesley, 2003. 63

[6] Carole Bernon, Valérie Camps, Marie-Pierre Gleizes, and Gauthier Picard. *Agent-Oriented Methodologies*, chapter Engineering Adaptive Multi-Agent Systems: the ADELFE Methodology, pages 172–202. In Henderson-Sellers and Giorgini [41], 2005. 90

[7] Rodney Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3):139–159, 1991. 49

[8] Sven Brueckner. *Return from the Ant — Synthetic Ecosystems for Manufacturing Control*. PhD thesis, Humboldt University, Berlin, Germany, 2000. 5

[9] Cristiano Castelfranchi. Mind as an Anticipatory Device: For a Theory of Expectations. In Massimo De Gregorio, Vito Di Maio, Maria Frucci, and Carlo Musio, editors, *BVAI*, volume 3704 of *Lecture Notes in Computer Science*, pages 258–276. Springer, 2005. 53

[10] Cristiano Castelfranchi and Emiliano Lorini. Cognitive Anatomy and Functions of Expectations. In *Cognitive Modeling of Agents and Multi-Agents Interactions*, pages 842–849, 2003. 53

[11] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves: Applying agent technology to support human organizations. In Haym Hirsh and Steve A. Chien, editors, *IAAI*, pages 51–58. AAAI, 2001. 2

[12] A. Chavez and Patie Maes. Kasbah: An agent marketplace for buying and selling goods. In *International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, 1996. 7, 11

[13] Caroline Chopinaud, Amal El Fallah-Seghrouchni, and Patrick Taillibert. Prevention of harmful behaviors within cognitive and autonomous agents. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, pages 205–209. IOS Press, 2006. 3, 7

[14] Caroline Chopinaud, Amal El Fallah-Seghrouchni, and Patrick Taillibert. Automatic generation of self-controlled autonomous agents. In Andrzej Skowron, Jean-Paul A. Barthès, Lakhmi C. Jain, Ron Sun, Pierre Morizet-Mahoudeaux, Jiming Liu, and Ning Zhong, editors, *IAT*, pages 755–758. IEEE Computer Society, 2005. 7

[15] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999. 106

[16] Corba website. `http://www.corba.org/`. 9

[17] CPN Tools website. `http://www.daimi.au.dk/CPNTools/`, Accessed in November 2006. University of Aarhus, Denmark. 105

[18] Chrysanthos Dellarocas. Toward Exception Handling Infrastructures in Component-based Software. In *Proceedings of the International Workshop on Component-based Software Engineering.*, 1998. 29

[19] Alexis Drogoul, B. Corbara, and S. Lalande. MANTA: New Experimental Results on the Emergence of (Artificial) Ant Societies. In N. Gilbert and R. Conte, editors, *Artificial Societies: The Computer Simulation of Social Life*, pages 190–211. UCL Press: London, 1995. 5, 64

[20] Marc Esteva, Bruno Rosell, Juan A. Rodríguez-Aguilar, and Josep Luís Arcos. Ameli: An agent-based middleware for electronic institutions. In *AAMAS*, pages 236–243. IEEE Computer Society, 2004. 9, 45

[21] Torsten Eymann, Boris Padovan, and Detlef Schoder. Avalanche - An Agent Based Value Chain Coordination Experiment. In *Workshop on Artificial Societies and Computational Markets (ASCMA'98) at Autonomous Agents '98*, pages 48–53, 1998. 26

[22] Jacques Ferber and Olivier Gutknecht. A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems. In *ICMAS*, pages 128–135. IEEE Computer Society, 1998. 37

[23] FIPA Agent Communication Language Specification. `http://www.fipa.org/repository/aclspecs.html`. Accessed in June 2006. 2, 6, 9, 34, 50, 55

[24] FIPA Agent Management Specification. `http://www.fipa.org/specs/fipa00023/SC00023K.html`. 9

[25] FIPA Abstract Architecture Specification. `http://www.fipa.org/specs/fipa00001/SC00001L.html`. 9

[26] Klaus Fischer, Jörg P. Müller, and Markus Pischel. A pragmatic BDI architecture. In Michael Wooldridge, Jörg P. Müller, and Milind Tambe, editors, *ATAL*, volume 1037 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 1995. 26

[27] Foundation for Intelligent Pysical Agents. FIPA Contract Net Interaction Protocol Specification. `http://www.fipa.org/specs/fipa00029/SC00029H.html`. Document number SC00029H, Accessed in October 2006. 12, 51

[28] Erich Gamma, R Helm, R Johnson, and J Vlissides. *Design Patterns*. Addison-Wesley, 1999. 5

[29] Alessandro Garcia, Holger Giese, Alexander Romanovsky, Ricardo Choren, Ho fung Leung, Carlos Lucena, Florian Klein, and Eric Platon. Software engineering for large-scale multi-agent systems - SELMAS 2006: Workshop report. *SIGSOFT Softw. Eng. Notes*, 31(5):24–32, 2006. 10

[30] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985. 6

[31] Maria Gini and Toru Ishida, editors. *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*. ACM, 2002. 100, 102

[32] Jacob Glazer and Ariel Rubinstein. Debates and Decisions, On a Rationale of Argumentation Rules. *Games and Economic Behavior*, 36:158–176, 2001. 7

[33] John B. Goodenough. Exception handling design issues. *SIGPLAN Not.*, 10(7):41–45, 1975. 18, 39

[34] John B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Commun. ACM*, 18(12):683–696, 1975. 3, 18, 39, 91

[35] John B. Goodenough. Structured exception handling. In *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 204–224, New York, NY, USA, 1975. ACM Press. 18, 39

[36] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, editors. *The Java™ Language Specification, Third Edition*. Addison-Wesley, 2005. 20

[37] Abdelkader Gouaïch, Fabien Michel, and Yves Guiraud. MIC*: A deployment environment for autonomous agents. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environment for Multi–Agent Systems'04*, volume 3374 of *Lecture Notes in Artificial Intelligence*, pages 109–126. Springer–Verlag, 2005. 7

[38] Zahia Guessoum, Nora Faci, and Jean-Pierre Briot. Adaptive replication of large-scale multi-agent systems - towards a fault-tolerant multi-agent platform. In Alessandro F. Garcia, Ricardo Choren, Carlos José Pereira de Lucena, Paolo Giorgini, Tom Holvoet, and Alexander B. Romanovsky, editors, *SELMAS*, volume 3914 of *Lecture Notes in Computer Science*, pages 238–253. Springer, 2005. 2

[39] Zahia Guessoum, Mikal Ziane, and Nora Faci. Monitoring and organizational-level adaptation of multi-agent systems. In *AAMAS*, pages 514–521. IEEE Computer Society, 2004. 2

[40] Staffan Hägg. A Sentinel Approach to Fault Handling in Multi-Agent Systems. In Chengqi Zhang and Dickson Lukose, editors, *Distributed AI*, volume 1286 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 1996. 33, 34, 68, 89

[41] Brian Henderson-Sellers and Paolo Giorgini, editors. *Agent-Oriented Methodologies*. Whitestein Series in Software Agent Technologies. Idea Group Publishing, 2005. 90, 95

[42] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999. 49

[43] Valérie Issarny. Concurrent Exception Handling. In Romanovsky et al. [85], pages 111–127. v, 4, 26, 27

[44] Valérie Issarny and Jean-Pierre Banâtre. Architecture-based Exception Handling. In *Hawaii International Conference on System Sciences*, 2001. 28

[45] Jadex Agent Platform Project. `http://sourceforge.net/projects/jadex`. Accessed in August 2006. 49

[46] Jason Agent Platform Project. `http://jason.sourceforge.net/`. Accessed in August 2006. 49

[47] Kurt Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis . In *Advances in Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416, 1991. 105

[48] Antonis C. Kakas, Paolo Mancarella, Fariba Sadri, Kostas Stathis, and Francesca Toni. The KGP model of agency. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 33–37. IOS Press, 2004. 44, 49, 64, 92, 94

[49] Gal A. Kaminka. *Execution Monitoring in Multi-Agent Environments*. PhD thesis, Computer Science Department—University of Southern California, 2000. 2, 6

[50] Gal A. Kaminka, David V. Pynadath, and Milind Tambe. Monitoring Teams by Overhearing: A Multi-Agent Plan-Recognition Approach. *Journal of Artificial Intelligence Research*, 17:83–135, 2002. 6

[51] Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In *Agents*, pages 62–68, 1999. 29, 33, 36, 45, 60, 91

[52] Mark Klein, P. Faratin, H. Sayama, and Y. Bar-Yam. Protocols for Negotiating Complex Contracts. *IEEE Intelligent Systems*, Nov./Dec.:32–38, 2003. 7

[53] Mark Klein, Juan A. Rodríguez-Aguilar, and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):179–189, 2003. 3, 15, 28, 29, 34, 42, 45, 89, 90

[54] Robert A. Kowalski and Marek J. Sergot. A Logic–Based Calculus of Events. *New Generation Computing*, 4:67–95, 1986. 30

[55] Nicholas Kushmerick. Software agents and their bodies. *Minds and Machines*, 7(2):227–247, 1997. 53

[56] François Legras and Catherine Tessier. Lotto: Group Formation by Overhearing in Large Teams. In *Autonomous Agents and Multi–Agent Systems*, pages 425–432. ACM Press, 2003. 3

[57] The MadKit Agent Platform Project. `http://www.madkit.org/`. Accessed in July 2006. 37

[58] Ashok U. Mallya. *Modeling and Enacting Business Processes via Commitment Protocols among Agents*. PhD thesis, North Carolina State University, Raleigh, United States, 2005. 35, 50

[59] Ashok U. Mallya and Munindar P. Singh. Modeling exceptions via commitment protocols. In *Autonomous Agents and Multi–Agent Systems*, pages 122–129, New York, NY, USA, 2005. ACM Press. 3, 35, 36, 44

[60] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the tota middleware. In *Proceedings of the International Conference On Pervasive Computing (Percom)*. IEEE CS Press, Orlando, Florida, USA, 2004. 9

[61] Hamza Mazouzi, Amal El Fallah-Seghrouchni, and Serge Haddad. Open protocol design for complex interactions in multi-agent systems. In Gini and Ishida [31], pages 517–526. 50, 105

[62] John McCarthy. Circumscription—A Form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13:27–39, 1980. Reprinted in [64]. 31

[63] John McCarthy. Applications of Circumscription to Formalizing Common Sense Knowledge. *Artificial Intelligence*, 28:89–116, 1986. Reprinted in [64]. 31

[64] John McCarthy. *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz.* Ablex, 1990. 100

[65] John McCarthy and Patrick J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969. 30

[66] Robert Miller and Anand Tripathi. The Guardian Model and Primitives for Exception Handling in Distributed Systems. *IEEE Trans. Software Eng.*, 30(12):1008–1022, 2004. 3, 22, 23, 45, 46

[67] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999. 51

[68] Tom Mitchell, Rich Caruana, Dayne Freitag, John McDermott, and David Zabowski. Experience with a learning personal assistant. *Communications of the ACM*, 37(7):81–91, 1994. 2

[69] Microsoft Win32 Structured Exception Handling for C++. `http://www.microsoft.com/msj/0197/Exception/Exception.aspx`. Accessed in October 2006. 20

[70] James Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, May-June 2002. 5

[71] H. Van Dyke Parunak. "Go to the Ant": Engineering Principles from Natural Multi-Agent Systems. *Annals of Operation Research*, 75:69–101, 1997. 5, 36

[72] H. Van Dyke Parunak. A survey of environments and mechanisms for human-human stigmergy. In Weyns et al. [108], pages 163–186. 36

[73] H. Van Dyke Parunak, Sven Brueckner, Mitch Fleischer, and James Odell. A design taxonomy of multi-agent interactions. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *AOSE*, volume 2935 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2003. 6

[74] H. Van Dyke Parunak, Sven A. Brueckner, Mitch Fleischer, and James Odell. A preliminary taxonomy of multi-agent interactions. In *Autonomous Agents and Multi–Agent Systems*, pages 1090–1091. ACM Press, 2003. 6

[75] H. Van Dyke Parunak and Danny Weyns, editors. *Autonomous Agents and Multi-Agent Systems, Special Issue on Environment for Multi-Agent Systems*, volume 14, number 1. Springer Netherlands, February 2007. 101, 103

[76] Eric Platon. Artificial intelligence in the environment: Smart environment for smarter agents in open e-markets. In *Proceedings of the Florida Artificial Intelligence Research Society*. AAAI, 2006. 11

[77] Eric Platon, Marco Mamei, Nicolas Sabouret, Shinichi Honiden, and H. Van Dyke Parunak. Mechanisms of the Environment for Mutli-Agent Systems, Survey and Opportunities. In *Autonomous Agents and Multi-Agent Systems* [75], pages 31–47. 6

[78] Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Overhearing and direct interactions: Point of view of an active environment. In Weyns et al. [108], pages 121–138. 6, 90

[79] Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Environment Support for Tag Interactions. In *Environment for Multi–Agent Systems*, 2006. 6, 90

[80] Anand S. Rao and Michael P. Georgeff. BDI Agents: From Theory to Practice. Technical report, Australian Artificial Intelligence Institute, 1995. 33, 64

[81] Mike Reddy and Gregory M.P. O'Hare. Blackboard systems: A survey of their application. *Artificial Intelligence Review*, May, 1991. 6

[82] Martin Rehák, Jan Tožička, Michal Pěchouček, Filip Železný, and Milan Rollo. An abstract architecture for computational reflection in multi-agent systems. In Andrzej Skowron, Jean-Paul A. Barthès, Lakhmi C. Jain, Ron Sun, Pierre Morizet-Mahoudeaux, Jiming Liu, and Ning Zhong, editors, *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 128–131. IEEE Computer Society, 2005. 60, 91

[83] Charles Rich and Candace L. Sidner. Collagen: When agents collaborate with people. In *Agents*, pages 284–291, 1997. 2

[84] Alexander B. Romanovsky. Exception Handling in Component-Based System Development. In *COMPSAC*, pages 580–598. IEEE Computer Society, 2001. 29, 30

[85] Alexander B. Romanovsky, Christophe Dony, Jørgen Lindskov Knudsen, and Anand Tripathi, editors. *Advances in Exception Handling Techniques (the book grow out of a ECOOP 2000 workshop)*, volume 2022 of *Lecture Notes in Computer Science*. Springer, 2001. 98, 103

[86] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Edition 2003. 49, 66

[87] Ken Satoh. An Application of Global Abduction to an Information Agent Which Modifies a Plan Upon Failure - Preliminary Report. In João Alexandre Leite and Paolo Torroni, editors, *CLIMA V*, volume 3487 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2004. 32, 94

[88] Ken Satoh and Keiji Yamamoto. Speculative computation with multi-agent belief revision. In Gini and Ishida [31], pages 897–904. 32, 94

[89] Peter Seibel. *Practical Common Lisp*. Apress, 2005. 20

[90] Shadows Project: Self-Healing Approach to Designing Complex Software Systems. `https://sysrun.haifa.il.ibm.com/shadows/`, Accessed in October 2006. 3

[91] Nazaraf Shah, Kuo-Ming Chao, Nick Godwin, Muhammad Younas, and Christopher Laing. Exception Diagnosis in Agent-Based Grid Computing. In *International Conference on Systems, Man and Cybernetics*, pages 3213–3219. IEEE, 2004. 34

[92] Jaime Simão Sichman, Rosaria Conte, Cristiano Castelfranchi, and Yves Demazeau. A social reasoning mechanism based on dependence networks. In *European Conference on Artificial Intelligence*, pages 188–192, 1994. 8

[93] Jaime Simão Sichman. DEPINT: Dependence-Based Coalition Formation in an Open Multi-Agent Scenario. *Journal of Artificial Societies and Social Simulation*, 1(2), 1998. 7

[94] Munindar P. Singh and Michael N. Huhns. *Service–Oriented Computing: Semantics, Processes, Agents*. Wiley, 2005. 1

[95] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. Computers*, 29(12):1104–1113, 1980. 12

[96] Frédéric Souchon, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Improving Exception Handling in Multi-agent Systems. In Carlos José Pereira de Lucena, Alessandro F. Garcia, Alexander B. Romanovsky, Jaelson Castro, and Paulo S. C. Alencar, editors, *SELMAS*, volume 2940 of *Lecture Notes in Computer Science*, pages 167–188. Springer, 2003. 37

[97] Squeak website. `http://www.squeak.org/`, Accessed in October 2006. 44

[98] Kostas Stathis, Wenjin Lu, Antonis C. Kakas, Neophytos Demetriou, Ulle Endriss, and Andrea Bracciali. PROSOCS: A platform for programming software agents in computational logic. In *From Agent Theory to Agent Implementation*, 2004. 49, 92

[99] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2000. 20

[100] Clemens Szyperski. *Component Software*. Addison-Wesley, 2002. 29

[101] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1994. 23, 45, 89

[102] Anand Tripathi and Robert Miller. Exception handling in agent-oriented systems. In Romanovsky et al. [85], pages 128–146. 22

[103] Maksim Tsvetovatyy, Maria L. Gini, Bamshad Mobasher, and Zbigniew Wieckowski. Magma: An agent based virtual market for electronic commerce. *Applied Artificial Intelligence*, 11(6):501–523, 1997. 7, 11

[104] Unified Modeling Language Specification, UML version 2.0. `http://www.omg.org/docs/formal/05-07-04.pdf`, August 2005. Accessed in December 2006. 12

[105] Javier Vázquez-Salceda. *The Role of Norms and Electronic Institutions in Multi-Agent Systems, The HARMONIA Framework*. Whitestein Series in Software Agent Technologies. Springer, 2004. 45

[106] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999. 1, 2

[107] Danny Weyns, Andrea Omicini, and James Odell. Environment, First-Order Abstraction in Multiagent Systems. In *Autonomous Agents and Multi-Agent Systems* [75], pages 5–30. 9, 40

[108] Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors. *Environments for Multi-Agent Systems II, Second International Workshop, E4MAS 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers*, volume 3830 of *Lecture Notes in Computer Science*. Springer, 2006. 101

[109] Danny Weyns, Kurt Schelfthout, Tom Holvoet, and Tom Lefever. Decentralized Control of E'GV Transportation Systems. In Franck Dignum, Sarit Kraus, and Munindar Singh, editors, *Autonomous Agents and Multi–Agent Systems*, pages 67–74. ACM Press, 2005. 5

[110] Danny Weyns, Elke Steegmans, and Tom Holvoet. Towards Active Perception in Situated Multi-Agent Systems. *Special Issue of the Journal on Applied Artificial Intelligence*, 18(8–9), 2004. 63

[111] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The michigan internet acutionbot: A configuarable auction server for human and software agents. In *Agents*, pages 301–308, 1998. 7, 11

[112] Jie Xu, Alexander B. Romanovsky, and Brian Randell. Coordinated Exception Handling in Distributed Object Systems: From Model to System Implementation. In *ICDCS*, pages 12–21, 1998. v, 4, 25, 26, 29, 46, 91

[113] Franco Zambonelli, Federico Bergenti, and Marie-Pierre Gleizes, editors. *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*. Kluwer Academic Publisher, 2004. 5

[114] Franco Zambonelli and H. Van Dyke Parunak. Signs of a Revolution in Computer Science and Software Engineering. In Paolo Petta, Robert Tolksdorf, and Franco Zambonelli, editors, *Engineering Societies in the Agent World*, volume 2577 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002. 8

[115] Roland Zimmermann. *Agent-based Supply Network Event Management*. Whitestein Series in Software Agent Technologies. Springer, 2006. 90

Analysis of the agent execution model

The agent execution model has been presented in chapter 4 as a framework that consists of specific data structures and algorithms. The aim of this appendix is to analyze properties of the model at a higher abstraction level: The flow of execution of the different algorithms is studied to verify systemic properties (e.g. liveness of activities) of the model, which is concretely a cycle of message processing and production.

Properties of the execution model

Automated tools were utilized to study the properties of the execution model. The model has been written as a Colored Petri Net and analyzed with CPNTools [47, 17]. The automation provided by this tool allowed to simulate and improve the execution model, and to exploit a model checker to verify high-level characteristics of the model, notably for deadlocks, liveness, and fairness issues along its execution. The development of additional convenience tools allowed to produce the information of this section.

The choice for the Colored Petri Net (CPN) representation was guided by several needs. The most important are the model of true concurrency and the convenient extension to composable formalisms. Concurrency matters are the subject of on-going research and do not appear explicitly in this document. CPN represent an 'investment' for future research including concurrency[2] inside the agent (an agent can be a multi-threaded application by itself). The composability is a weakness of standards Colored Petri Nets, but equivalent formalism such as Hierarchical CPN or Recursive Hierarchical CPN are possible extensions that palliate this weakness (see [61] for a brief survey and references). The composability is an important property of the formal model, in order to compose the execution model with protocols and handlers represented as CPN as well [61].

[2]The execution model already manages consistently the concurrent execution of several protocols. The concurrency in this section refers to having the agent as a multi-threaded process.

Mapping to a Colored Petri Net

Fig. 1 shows the whole execution model as a Colored Petri Net. The simulation and model checking are executed under some hypothesis to study the properties. The hypothesis are exposed hereafter as heuristics to optimize the simulation and verification of the model. They do not diminish the result of the analysis. The network uses Standard ML expressions for the syntax and execution of variables and functions.

The transitions of the network use full names and the mapping to the execution model is straightforward. Places mostly contain abbreviations of the corresponding states to avoid clutter on the figure. The following table 1 develops the abbreviations.

The initial marking allows to run the network in an infinite processing loop. The tokens on Init and Out trigger the agent perception. After firing the perception transition, a new token is immediately put on Init to prepare the next perception of the agent. The next input will occur whenever the Out place receives a token, either when the agent outputs a message or when an input is ignored (this mechanism allows to simulate a continuous execution). The perception transition produces a random 'message' that represents an ACL message. The messages have a simple pattern with essential information (sender, receiver, content), expressed in Standard ML, and extended with random information about the nature of the message. If the message passes the relevance and expectation filters, the kind of exception is predefined to reduce the complexity of the model, without impairing its semantics. A message is then defined as a record color set (keyword `colset`).

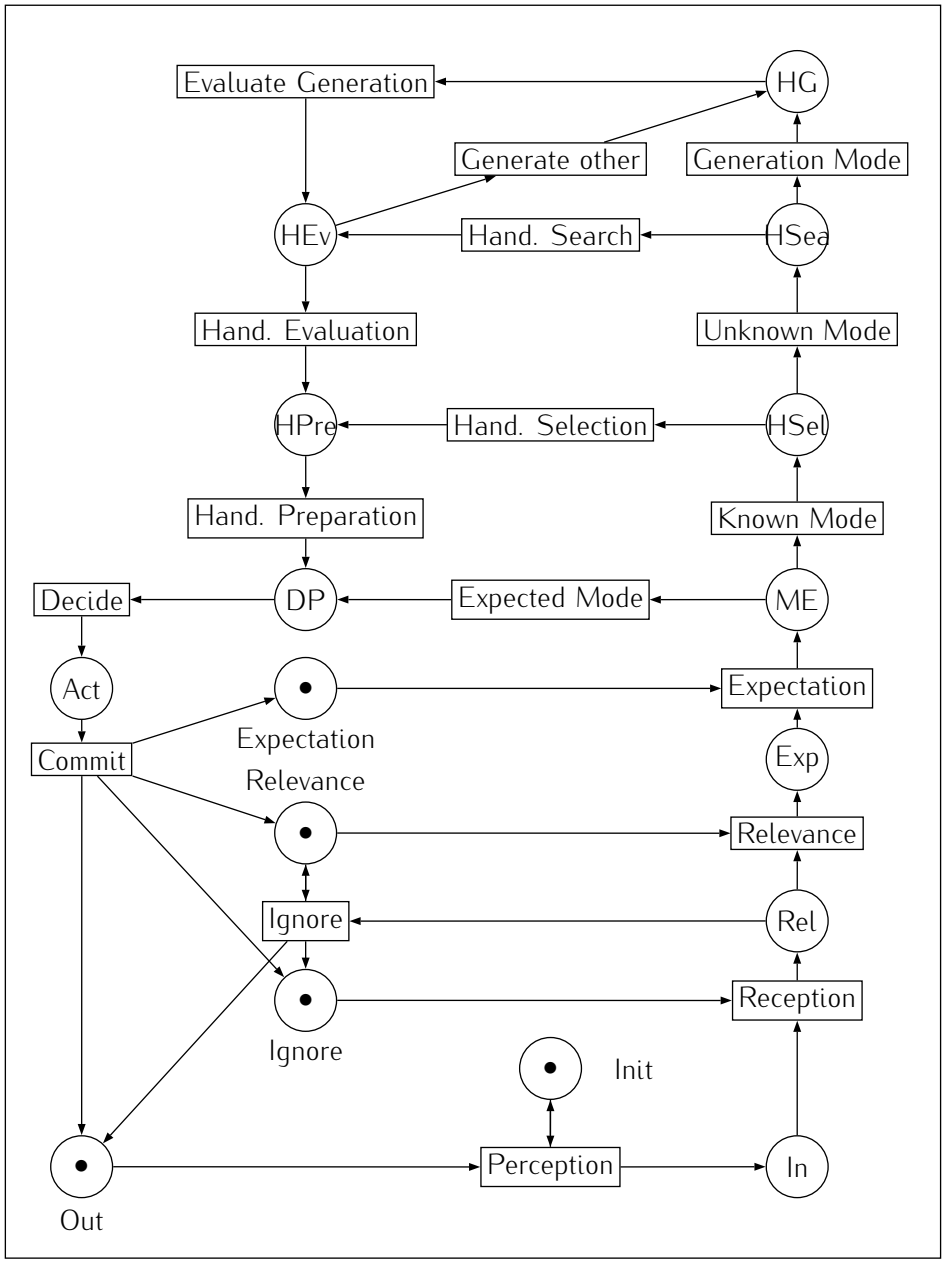
```
colset Msg = record from:STRING * to:STRING * content:STRING *
                    sel:BOOL * sea:BOOL * eva:BOOL;
```

The message is a record with similar field names as for the formal model. In addition, `sel` is a boolean that, if true, states that the message will have a handler available, none otherwise. Similarly, `sea` pre-define the success or failure for the handler search, and `eva` for the handler evaluation. The usual fields of the message are each a single character string to reduce the complexity of the state space analysis, which is the common abstraction method in model checking [15].

The message is then forwarded to the reception transition, which consumes both tokens on In and on Ignore. The token on Ignore serves to continue the execution to the next message when the current message is not relevant. The message is then tested for relevance on the corresponding transition. The following function was written to compare the message to the relevance criteria, which is also a single character string.

```
fun matchRel(r:RelevanceCriteria,p:Msg):BOOL=
  (* init *)
  if (#value r = init)
  then
```

Figure 1: Execution model of agent with exception management capabilities: Formalization in a Colored Petri Net




```
  true
else
  (* if there is any match, it is relevant *)
  if(#value r = #dest p orelse #value r = #content p)
  then
```

Abbreviation	Full name
In	Input of the execution model (Message from the environment)
Out	Output of the execution model (Message to the environment)
Rel	Test relevance
Exp	Test expectation
ME	Message to evaluate
HSel	Handler to select
HSea	Handler to search
HG	Handler to generate
HEv	Handler to evaluate
HPre	Handler to prepare
DP	Decision to process
Act	Action to commit

Table 1: Full name of places on the CPN

```
    true
  else
    false;
```

The matching relevance algorithm is simply to check whether the recipient of the message or the content matters to the agent. If any of them matches the relevance criteria, the function returns `true` to express the message is relevant, and `false` otherwise. If the message is relevant, it is forwarded to the Exp place. Otherwise, the token on 'Relevance' is not consumed by the relevance transition, but by the ignore transition that puts a new token on the Ignore and Out places to process the next input.

The expectation matching occurs in the same way as relevance, and the message is forwarded with an indicator variable, either 'expected' or 'unexpected'. Expected messages are forwarded to DP and the decide transition to generate the action to commit in the environment and a pair of new relevance and expectations for the next cycle. A token is also placed on the 'Ignore' place to allow the next input in this successful process of a message.

Unexpected messages are forwarded upward to Known exception mode. A handler selection is attempted according to the pre-defined information in the message. Success of the handler selection leads to the preparation and then back to the DP place. Unsuccessful selection passes the message to handler search. The message is sent to evaluation along with a handler if the search is successful, and to handler generation in the contrary case. At the evaluation stage, the message and handler are sent to the preparation place if the evaluation is positive, or to generation for a better handler in HG. The generation always succeeds to produce a handler (the evaluate generation transition is always true to simulate the production of a default handler at least), so that the execution is guaranteed to reach and pass the eval-

uation eventually. Once the evaluation is positive, the message and handler are prepared and the execution continues with the DP decision process of the agent.

Analysis of the model.

The analysis of the model has been conducted through simulations and model checking. The simulation produces log files as traces, but CPNTools also provides animations of the network to observe the evolution of the marking.

Several runs of the simulation have never ended on either a deadlock or liveness issue. The simulations do not allow to conclude however that the network is safe and starvation-free. Model checking is one technique that allows a comprehensive exploration of the state space. The following reports are the results for deadlock, liveness, and fairness analysis. A deadlock in the execution model means that the execution will stop in a state that is not a terminal state, i.e. no transition can fire anymore. As the model is designed to continue infinitely, it must contain no deadlock. Deadlocks must be avoided to show that the execution can always evolve and remain in states decided in the model. Liveness issues occur whenever some transitions of the model cannot be fired at all or from some point in the execution. In other words, liveness issues means that parts of the model cannot be used anymore. Liveness issues must be avoided to guarantee that the agent maintains all its functionalities, represented by the successive boxes in the execution model. Fairness is related to a 'fair choice' of the agent functionalities, which means that any functionality is eventually executed if the agent runs infinitely. Fairness issues occur whenever some transitions execute 'infinitely more often' than others. A practical consequence of fairness issues is that a subset of transitions execute, whereas others never fire. The difference with liveness issues is that all transitions can potentially fire when there is a fairness problem, even though the problem cause partial ones to take all opportunities to fire, thus blocking others.

The results of the property verification presented in Fig. 2 state that the execution model has no deadlock or liveness issues. This first result means that agents implementing the execution model can run infinitely without encountering problems due to the model, and they can exploit all the model functionalities along any run. The results also show that most but two transitions are fair. The two partial transitions are the `Perception` and `Reception` at the bottom right of the network in Fig. 1. As observed during simulations of the network, these two transitions fire significantly more often than more others. Messages are created as tokens by the Init and Out places, thus necessarily firing the two transitions. Only one type of message can however pass the `Relevance` transition according to the model. That is, the message token must match the relevance criteria token on the relevance place to be further processed by the agent. All message tokens that do not match the relevance criteria are consumed by the `Ignore` transition and a new message token is created that immediately enables `Perception`. The series `Perception-Reception` is therefore triggered significantly more often than any other transition. We could evaluate that they fire twice as often as others on aver-

```
CPN Tools state space report for: ExecutionModel.cpn

 Liveness Properties
----------------------------------------------------
  Dead Markings
     None
  Dead Transition Instances
     None
  Live Transition Instances
     All

 Fairness Properties
----------------------------------------------------
  Act 1
     Fair
  EvaluateGeneration 1
     Fair
  Expectation 1
     Fair
  GenerationMode 1
     Fair
  GenerateOther 1
     Fair
  Hand.Search 1
     Fair
  Hand.Evaluation 1
     Fair
  Hand.Preparation 1
     Fair
  Hand.Selection 1
     Fair
  Ignore 1
     Fair
  KnownMode 1
     Fair
  ExpectedMode 1
     Fair
  Decide 1
     Fair
  Perception 1
     Impartial
  Reception 1
     Impartial
  Relevance 1
     Fair
  UnknownMode 1
     Fair
```

Figure 2: Output of the automated verification tools

age. The probability that the message matches the relevance criteria is 33.3% for each cycle of the agent due to the simulation settings. The two transitions execute consequently 66.6% of the cycles, whereas other transitions can run only 33.3% of the time.

Conclusion

The execution model presented in this section describes how agents can embed individual mechanisms that are suitable in managing exceptional situations. The different mechanisms are in place so that agents can automatically leverage handlers provided by the designers. The analysis of the model shows it is deadlock-free and alive for all its transitions, which proves that the agent can react to any well-formed input and maintain its functionalities available over time. The fairness issue shows that the input functionality of the agent filters out a majority of messages and may prevent the agent to execute. This phenomenon is not an issue in the present case and becomes a property of the model, since the filtering has been introduced so that agents process only meaningful messages. In other words, the agent can focus on messages of interest and execution cycles are saved due to unfair property of the `Perception` and `Reception` transitions. This filtering is indeed essential when agents are deployed in unknown environments, where relevant information must be identified to avoid wasting computation time on useless percepts.

The mechanisms introduced in the execution model form the application-independent part of our exception management approach. The next section is devoted to consider the execution model in the perspective of architecting agents, so that development can rely on a predefined architecture of the agent software and concentrate on the application-dependent part of the code. In particular, handling depends generally on the application at hand (handling differs in managing a stock of food or furniture), and the next chapter studies the modeling of handlers to provide the execution model with appropriate functionalities.

Publications

List of publications in relation with the thesis and related work.

1. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. An Architecture for Exception Management in Multi-Agent Systems. Paolo Giorgini and Brian Henderson-Sellers, editors. *International Journal on Agent-Oriented Software Engineering*, to appear. Indersciences, 2007.

2. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. A Definition of Exceptions in Agent-Oriented Computing. Gregory O'Hare, Michael O'Grady, Oguz Dikenelli, and Alessandro Ricci, editors. *Engineering Societies in the Agent World, Seventh International Workshop, ESAW 2006, Dublin, Ireland, September 2006, Selected Revised and Invited Papers*, volume 4457 of *Lecture Notes in Computer Science*. Springer, 2007.

3. Eric Platon, Marco Mamei, Nicolas Sabouret, Shinichi Honiden, and H. Van Dyke Parunak. Mechanisms of the Environment for Multi-Agent Systems, Survey and Opportunities. H. Van Dyke Parunak and Danny Weyns, editors. *Autonomous Agents and Multi-Agent Systems, Special Issue on Environment for Multi-Agent Systems*, volume 14, number 1. Springer Netherlands, February 2007, pages 31–47.

4. Alessandro Garcia, Holger Giese, Alexander Romanovsky, Ricardo Choren, Ho fung Leung, Carlos Lucena, Florian Klein, and Eric Platon. Software engineering for large-scale multi-agent systems - SELMAS 2006: Workshop report. *SIGSOFT Softw. Eng. Notes*, 31(5):24–32, 2006.

5. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Challenges in Exception Handling for Multi-Agent Systems. Ricardo Choren, Alessandro Garcia, Holger Giese, Ho-fung Leung, Carlos Lucena, and Alexander Romanovsky, editors. *Software Engineering for Large-Scale Multi-Agent Systems, Fifth International Workshop, SELMAS 2006, Shanghai, China, May 22-23, 2006, Selected Revised and Invited Papers*, *Lecture Notes in Computer Science*. Springer, 2007.

6. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Environment Support for Tag Interactions. Danny Weyns, H. Van Dyke Parunak, and Fabien

Michel, editors. *Environments for Multi-Agent Systems III, Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers*, *Lecture Notes in Computer Science*. Springer, 2007.

7. Eric Platon. Artificial intelligence in the environment: Smart environment for smarter agents in open e-markets. In *Proceedings of the Florida Artificial Intelligence Research Society*. AAAI, 2006.

8. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Tag Interactions in Multi-Agent Systems: Environment Support. Marie-Pierre Gleizes, Gal A. Kaminka, and Sascha Ossowski, editors. *European Workshop on Multi-Agent Systems, EUMAS 2005*, Bruxelles, Belgium, December 2005.

9. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Oversensing with a softbody in the environment - Another dimension of observation. Gal A. Kaminka, David V. Pynadath, and Christopher W. Geib, editors. *Modeling Others from Observation, Second International IJCAI Workshop, MOO 2005*, Edinburgh, Scotland, July 30, 2005.

10. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Overhearing and direct interactions: Point of view of an active environment. Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors. *Environments for Multi-Agent Systems II, Second International Workshop, E4MAS 2005, Utrecht, The Netherlands, July 25, 2005, Selected Revised and Invited Papers*, volume 3830 of *Lecture Notes in Computer Science*, pages 121–138. Springer, 2006.

11. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Modeling Interactions in Assistant Teams. In *Proceedings of the International Conference on Active Media Technology, AMT 2005*, Takamatsu, Japan. IEEE, 2005.

12. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. T-compound: An Agent-Specific Design Pattern and its Environment. Cesar Gonzalez-Perez and Brian Henderson-Sellers, editors. *Agent-Oriented Methodologies, Third International OOPSLA Workshop, AOM 2004*, Vancouver, Canada, October 2004, COTAR Publications, 2004.

13. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. T-compound Interaction and Overhearing Agents. Marie Pierre Gleizes, Andrea Omicini, and Franco Zambonelli, editors. *Engineering Societies in the Agents World V, Fifth International Workshop, ESAW 2004, Toulouse, France, October 2004, Selected Revised and Invited Papers*, volume 3451 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2004.

14. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Introducing Participative Personal Assistant Teams in Negotiation Support Systems. Michael

Wayne Barley and Nik Kasabov, editors. *Intelligent Agents and Multi-Agent Systems, Seventh Pacific Rim International Workshop on Multi-Agents, PRIMA 2004, Auckland, New Zealand, August 2004, Selected Revised and Invited Papers*, volume 3371 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2004.

Index