Sujet de la thèse:

Modélisation de la gestion des exceptions dans les systèmes multiagents.

Soutenue le 25 juin 2007

Devant le jury composé de:

Pr. Olivier Boissier       Rapporteur
Pr. Amal El-Fallah Seghrouchni  Directeur de thèse
Pr. Shinichi Honiden       Examinateur
Pr. Gal A. Kaminka          Rapporteur
Pr. Jean-Luc Koning         Examinateur
Pr. Jacques Malenfant       Examinateur
Dr. Nicolas Sabouret        Directeur de thèse
MODELING EXCEPTION MANAGEMENT
IN
MULTI-AGENT SYSTEMS

by Eric Platon

In Partial Fulfillment of the
Requirements for the Grade of
Doctor of Philosophy in Computer Science

Laboratoire d’informatique de Paris 6,
Université Pierre et Marie Curie

2007
Résumé de la thèse

Les systèmes multiagents sont souvent présentés comme une étape majeure dans l'évolution des techniques de développement logiciel. Ils sont reconnus comme particulièrement adaptés à la conception de systèmes complexes et distribués, ce que les applications logicielles modernes sont dans la plupart des cas pour répondre aux besoins actuels et à venir. Les systèmes multiagents (SMA) forment une classe de systèmes distribués, constitués d'entités logicielles autonomes et interactives nommées agents. Ils héritent et étendent les techniques de génie logiciel et d'intelligence artificielle dans le but de proposer des modèles et techniques adaptées aux problématiques actuelles des systèmes informatiques. Les approches SMA visent les systèmes à grande échelle (il est coutume de parler de « société » d'agents), ouverts (le nombre d'agents peut évoluer au cours de l'exécution) et hétérogènes (les agents peuvent être conçus par des méthodes différentes). La recherche actuelle se porte sur des méthodes de collaboration entre agents pour la réalisation de tâches d'une manière hautement modulaire et flexible. Les domaines d'application des SMA sont variés, de telle sorte qu'on les retrouve dans des robots autonomes déployés dans des environnements réels comme des usines automatisées, ou bien dans des assistants logiciels intelligents qui supportent l'utilisateur dans ses activités sur Internet.

Les SMA sont cependant des logiciels avant tout, et les techniques de développement traditionnelles ont démontré que la construction de systèmes fiables, robustes et résilients demande des efforts et pratiques appropriés. L'étude de telles propriétés dans les méthodes traditionnelles fait référence au terme anglais « dependability », c'est-à-dire le degré de confiance que l'utilisateur porte au fonctionnement d'un système informatique. Cette confiance crée un lien de dépendance de l'utilisateur au système que des techniques logicielles cherchent à assurer et augmenter. En particulier, les techniques de tolérance aux fautes ont permis d'améliorer la qualité des logiciels dans des cas de systèmes fermés et homogènes, avec de rares cas de systèmes ouverts. Les techniques basées sur les SMA permettent des solutions adaptées à des systèmes ouverts et hétérogènes, comme la demande actuelle en logiciel et en infrastructure le requiert aujourd'hui.

Parmi les techniques logicielles pour améliorer la fiabilité, la robustesse et la résilience des systèmes, le traitement des exceptions est connu pour sa généralité et sa simplicité. Les langages de programmation intègrent depuis de nombreuses années des mécanismes pour gérer de manière pratique et systématique des conditions exceptionnelles lors de l'exécution du programme. Les recherches en informatique distribuée, architectures logicielles et à base de composants ont cependant démontré que le traitement des exceptions nécessite dans de tels cas des mécanismes particuliers. Les SMA mettent en avant des propriétés difficiles à réaliser qui mènent à reconsidérer cette question dans son ensemble.

L'objectif de ce travail de thèse est d'étudier la notion d'exception de fonctionnement dans les SMA et de proposer un cadre de conception et d'implémentation adapté à leurs propriétés d'ouverture, d'hétérogénéité et plus particulièrement d'autonomie de leurs agents. Les travaux existants à ce sujet dans la littérature ont permis d'obtenir des résultats en ce qui concerne la gestion des exceptions « au niveau système », ce qui signifie que les mécanismes proposés sont extérieurs aux agents et visent une surveillance globale des activités au sein du système. Ces approches s'appliquent à des cas particuliers de SMA où les agents ne peuvent pas conserver leur autonomie et sont supposés parfaitement coopératifs lors du traitement de conditions exceptionnelles, ce qui n'est pas vérifiable en général. Dans ce travail de thèse, une approche est proposée « au niveau agent », la capacité d'un agent de gérer des situations exceptionnelles est vue comme un pré-requis.
à sa robustesse et à la conservation de son autonomie. Les agents conservent continûment
au cours de leur exécution la capacité de choisir quand initier un traitement d'exception et
quand accepter des supports extérieurs tels que les mécanismes au niveau système. Cette
approche est complémentaire avec les techniques existantes, de telle sorte qu'elle participe
tà l'amélioration des qualités de fiabilité, robustesse et résilience des SMA.

L'approche développée dans cette thèse repose sur un modèle d'exécution des agents
qui garantit que chacun d'entre-eux conserve le contrôle de son exécution, même en cas
de conditions exceptionnelles. Le modèle permet à l'agent de décider si un événement
doit être considéré comme une exception d'après un ensemble de prédictions générées
automatiquement et exactement sur les futurs événements attendus par l'agent, ce qui
est un choix subjectif et dépendant du contexte d'exécution de cet agent. Ce modèle
est décrit formellement et accompagné d'une architecture logicielle correspondante pour
faciliter ses implémentations, et une méthode d'utilisation conjointe avec les approches au
niveau système nommée Uni. Cette architecture est appliquée à un système complet à titre
d'exemple. Cette application permet de comparer le modèle à d'autres approches existantes
et d'évaluer son coût à l'exécution. Les perspectives de ce travail de thèse poursuivent la
lignée de problématiques soulevées par les besoins en fiabilité, robustesse et résilience des
SMA. En particulier, la génération automatique de méthodes de traitement des exceptions
dans des situations génériques est une extension dont l'intérêt réside dans l'augmentation
potentielle de l'autonomie des agents.
Multi-agent systems are often presented as the next major approach to engineer software, in the context of increasing complexity of 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 required mechanisms related to 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 an original 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 addressed in the framework proposed by this thesis to develop the dependability of MAS. In particular, the automatic generation of handling strategies by agents in a range of situations is a promising capability relying on AI techniques that can expand the autonomy of agents in dealing with various exceptional situations.
Contents

List of Figures ix
List of Tables xii
Acknowledgments xiii

1 Scope and Goals 1
  1.1 Concepts in Multi-agent systems .......................... 5
  1.2 Purpose & Scope of this document .......................... 10
  1.3 Case study ............................................. 11
  1.4 Organization .......................................... 15

I Agent Exceptions and Management by Autonomous Entities 17
  2 Related Work ............................................. 19
      2.1 Exceptions in programming languages ................... 19
          2.1.1 The original semantics of exception handling .... 20
          2.1.2 Alternative model: Condition handling in LISP .... 23
          2.1.3 Programming exceptions and agents ................. 23
      2.2 Exceptions in Distributed systems ...................... 24
          2.2.1 The Guardian ...................................... 24
          2.2.2 Coordinated and cooperation exception handling in distributed objects ........................................... 27
      2.3 Architecture-level and Component-level exceptions .... 30
          2.3.1 Architecture-level exception handling ............... 30
          2.3.2 Exceptions in Component-based Software Development ... 30
      2.4 Exception in Logics .................................... 32
          2.4.1 Default logic and Circumscription .................. 32
          2.4.2 Abductive reasoning .................................. 34
      2.5 Exceptions in MAS research ............................ 35
          2.5.1 The sentinel-based architecture ..................... 35
          2.5.2 Reliability database and sentinel-like agents ...... 36
2.5.3 Agent exceptions in commitment protocols ....................... 36
2.5.4 On-line execution monitoring .................................................. 38
2.5.5 Stigmergic systems ................................................................. 38
2.5.6 SaGE in the MadKit platform ................................................. 39
26 Survey conclusion ................................................................. 40

3 Definition ................................................................. 41
3.1 Agent exception .......................................................... 42
3.2 Programming and agent exceptions ............................................ 43
  3.2.1 From programming to agent exceptions ......................................... 44
  3.2.2 From agent to programming exceptions ......................................... 45
33 Exception space in Multi-agent systems ........................................ 45
34 Revisiting the terminology on exception management ..................... 48
35 Conclusion ................................................................. 49

II Robust Agent Execution Model and Engineering Framework 51

4 Agent Execution Model ............................................................... 53
  4.1 Assumptions and focus ....................................................... 53
  4.2 Syntax of the agent execution model .......................................... 55
    4.2.1 Literals and formulas ..................................................... 55
    4.2.2 Predicates .................................................................. 56
  4.3 Semantics of the agent execution model ......................................... 60
    4.3.1 Preliminary semantics: Plan and protocol operations ................... 60
    4.3.2 Agent state .............................................................. 60
    4.3.3 Execution procedures .................................................... 61

5 Extensions for more robust agents .................................................. 75
  5.1 Extension of the execution cycle .................................................. 76
  5.2 Acquisition of additional handling actions & discussion on generation mechanisms .................................................. 77
    5.2.1 Acquisition of an additional handler ....................................... 77
    5.2.2 Discussion on handler generation .......................................... 79
  5.3 Automated evaluation of handling protocols ..................................... 80
    5.3.1 Running Example ........................................................... 80
    5.3.2 Formalization of the problem ................................................. 82
    5.3.3 Handler evaluation .......................................................... 84
    5.3.4 Discussion .................................................................. 87
  5.4 Complexity analysis .............................................................. 88
    5.4.1 Notations .................................................................. 88
    5.4.2 Complexities .............................................................. 89
# Engineering framework

## 6.1 Framework

- Services of the framework
- Design tasks

## 6.2 Software Architecture

- Abstract architecture
- Elements of the architecture
- Correspondence table with the execution model

# Experiments and Validation

## 7.1 Experimental settings

- Scope of the EMS implementation
- Experimental protocol
- Technical details

## 7.2 Qualitative Analysis and Comparison

- Quality criteria
- Comparison

## 7.3 Experimental Results

- Overhead cost of exception management mechanism: Exception-free and EMS versions of the system
- Comparison between the Plain and EMS exception management systems

## 7.4 Conclusion

# Uniφ: Building Agent Systems Robust to Exceptions

## 8 Exception Management with Autonomy-Ready Sentinels

- Tag interactions and softbody
- Rationale of the interaction model
- Formalization
- Uniφ: Sentinels and autonomous agents
- Description of the MAS Uniφ framework
- Environmental contract protocol

## 9 Example system with Uniφ

- Scenario of an agent death in the case study
- Approach based on Autonomy-Ready Sentinels
- Execution example
- Discussion
10 Conclusion ........................................ 141
  10.1 General contributions of the present work ............. 142
  10.2 Contributions to Agent-Based Software Engineering .... 142
  10.3 Contributions to Distributed Artificial Intelligence .... 143
  10.4 Future perspectives ................................... 144

Bibliography ........................................ 145

Analysis of the agent execution model ......................... 157

Publications ........................................ 165
List of Figures

1.1 Basic exception handling in programming languages .................. 4
1.2 Picture for an agent throughout the chapters ...................... 5
1.3 Dependency network among actors on the market .................. 12
1.4 Contract net protocol in the market ............................... 13

2.1 Sample Java code to illustrate syntactic units and handler search . 21
2.2 State-chart describing the operational semantics of exception handling in many programming languages .......................... 22
2.3 State-chart describing the operational semantics of exception handling with the ‘guardian’ ........................................... 25
2.4 Architecture of the ‘primary-backup’ fault tolerance .................. 26
2.5 Architecture of the ‘primary-backup’ with the guardian .............. 26
2.6 The coordinated atomic action model ................................ 27
2.7 The exception graph model (reproduced from [135]) .................. 28
2.8 Syntactic bind at compile-time in a multi-procedure (reproduced from [52]) 29
2.9 Hierarchy organization of a MAS ................................ 34
2.10 Representation of the original sentinel approach .................... 35
2.11 Reliability database in the sentinel approach ...................... 37

3.1 View on the semantics of programming exceptions: Exceptions are decided on the operation side ................................. 42
3.2 Target of the semantics of agent exceptions: Exceptions are decided by agents themselves ............................................. 42
3.3 Agents and their exception levels ................................... 44
3.4 Relational mapping in an abstract exception space: Programming exceptions can breed agent exceptions, but not conversely .......... 46

4.1 Nested structure of the agent execution contexts .................... 62
4.2 Relationship between the different operators depending on their level in the agent architecture. Arrows are ‘calls’ to the target procedure ............................. 63

5.1 Execution cycle of an agent with incremental exception handling mechanisms ....................................................... 76
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Order protocol between client and store</td>
<td>81</td>
</tr>
<tr>
<td>5.3</td>
<td>Representation of the SOP, for the client</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>Enactment of the SOP with an exception</td>
<td>84</td>
</tr>
<tr>
<td>5.5</td>
<td>Possible Handlers from $S_x$ to $S_4$</td>
<td>84</td>
</tr>
<tr>
<td>5.6</td>
<td>Preference tree over constraints and scoring</td>
<td>85</td>
</tr>
<tr>
<td>6.1</td>
<td>Agent base architecture for exception management</td>
<td>94</td>
</tr>
<tr>
<td>7.1</td>
<td>Coverage of the implementation (plain lines) over the execution model (plain and dashed lines)</td>
<td>98</td>
</tr>
<tr>
<td>7.2</td>
<td>Experimental protocol of the experiments</td>
<td>100</td>
</tr>
<tr>
<td>7.3</td>
<td>Sample code from the experiments: Agent method <code>handlerSelection</code></td>
<td>102</td>
</tr>
<tr>
<td>7.4</td>
<td>Number of execution cycles completed by agent 'Machine Assembler 1' - No EMS</td>
<td>105</td>
</tr>
<tr>
<td>7.5</td>
<td>Number of execution cycles completed by agent 'Machine Assembler 1' (red) - With EMS</td>
<td>106</td>
</tr>
<tr>
<td>7.6</td>
<td>Average number of execution cycles completed by agents over 100ms periods - Exception-free</td>
<td>107</td>
</tr>
<tr>
<td>7.7</td>
<td>Average number of execution cycles completed by agents over 100ms periods - EMS</td>
<td>108</td>
</tr>
<tr>
<td>7.8</td>
<td>Capital of agent 'Machine Assembler 1' over time (red), and Bezier approximation (green) - No EMS</td>
<td>110</td>
</tr>
<tr>
<td>7.9</td>
<td>Capital of agent 'Machine Assembler 1' over time (red), and Bezier approximation (green) - With EMS</td>
<td>110</td>
</tr>
<tr>
<td>7.10</td>
<td>Average of the capitals of agents over time (red), and Bezier approximation for the average and each agent (other colors) - No EMS</td>
<td>111</td>
</tr>
<tr>
<td>7.11</td>
<td>Average of the capitals of agents over time (red), and Bezier approximation for the average and each agent (other colors) - With EMS</td>
<td>111</td>
</tr>
<tr>
<td>7.12</td>
<td>Average number of exceptional situations in the agent activities over time (red), and number of exceptions recognized by each agent (other colors) - With EMS</td>
<td>112</td>
</tr>
<tr>
<td>7.13</td>
<td>Evolution of the capital of agents and exception occurrences</td>
<td>113</td>
</tr>
<tr>
<td>7.14</td>
<td>Average number of relevance rules generated by agents over time (red), and Bezier approximation for the average and each agent (other colors) - With EMS</td>
<td>114</td>
</tr>
<tr>
<td>7.15</td>
<td>Average number of expectations rules generated by agents over time (red), and Bezier approximation for the average and each agent (other colors) - With EMS</td>
<td>115</td>
</tr>
<tr>
<td>8.1</td>
<td>Unifi approach as a complementarity of the agent- and system-level exception management systems</td>
<td>122</td>
</tr>
<tr>
<td>8.2</td>
<td>Left: Tag monitoring interaction. Right: Tag fortuitous interaction</td>
<td>125</td>
</tr>
<tr>
<td>8.3</td>
<td>Environment validates and commits the influence</td>
<td>129</td>
</tr>
<tr>
<td>8.4</td>
<td>Environment prevents the influence</td>
<td>129</td>
</tr>
</tbody>
</table>
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.5</td>
<td>Environmental effect on the agent public state.</td>
<td>130</td>
</tr>
<tr>
<td>8.6</td>
<td>Public State Management from left to right: the top-left agent modifies its public state; the environment validates the change; the change is spread to neighbors</td>
<td>131</td>
</tr>
<tr>
<td>8.7</td>
<td>Model of Multi-Agent System</td>
<td>132</td>
</tr>
<tr>
<td>8.8</td>
<td>Environmental contract protocol</td>
<td>133</td>
</tr>
<tr>
<td>9.1</td>
<td>Agents and sentinels in the Uniφ-based version of the consortium</td>
<td>136</td>
</tr>
<tr>
<td>9.2</td>
<td>Sequence of partial system states in the considered run (passive observation)</td>
<td>138</td>
</tr>
<tr>
<td>1</td>
<td>Execution model of agent with exception management capabilities: Formalization in a Colored Petri Net</td>
<td>159</td>
</tr>
<tr>
<td>2</td>
<td>Output of the automated verification tools</td>
<td>162</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Exception space of agents: 6 classes of exception ..................... 46
4.1 Literals in the model ........................................ 55
4.2 Reference table of the execution procedure .......................... 63
5.1 Structure evaluation ........................................ 85
5.2 Evaluation of the handlers in the example ........................... 87
5.3 Cost table depending on the execution type .......................... 89
6.1 Tasks for designing agents with exception management capabilities ... 93
6.2 Correspondence table between the execution model and the architec-
tural elements, according to Fig. 5.1 and Fig. 6.1 .................... 96
7.1 Qualitative comparison ........................................ 103
7.2 Comparison of the performance characteristics ..................... 107
7.3 Evaluation of the theoretical complexity ........................... 109
7.4 Comparison of the performance characteristics ..................... 116
7.5 Average computational cost of an agent cycle in terms of execution time 116
1 Full name of places on the CPN ........................................ 160
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 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 would like to particularly thank the members of my jury, who honored the time spent on this document by spending time themselves on its pages and the other activities relative to the final stages of the Ph.D. course. Pr. Olivier Boissier, Dr. Gal A. Kaminka, Pr. Amal El-Fallah Seghrouchni, and Pr. Jean-Luc Koning have contributed to this work, but also inspired it from my study of their research.

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 understandings—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 to 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 despite 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, Alessandro Ricci, Andrea Omicini, Alessandro Garcia, 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 support.

Eric Platon, Tokyo, April 2006.
Multi-agent systems consist of software programs named agents that execute in parallel and interact to achieve functions of the system in an environment [127, 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 [113].

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’ [83, 101], 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 teamwork [12].

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 [27]. 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’ [127, 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.

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 [59, 47, 46]. 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 generally
‘autonomic computing’ [3, 109]. Autonomous agents are expected to be resilient, i.e. to accomplish their 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 [82, 68, 64, respectively]. However, micro-approaches have much less achievements, despite the potential of having agents really resilient with respect to their autonomy. Little engineering results have been obtained, such as work on self-organizing systems, self-controlled agents, and commitment protocols [88, 14, 73, 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 [64].

**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 [41]. 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.
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 [135, 52]. 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 were 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 [88, 86]. The manipulation through the chapters 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., [34, 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 [21, 88, 11] to complete software systems [136, 132], 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

\(^1\)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. 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) [27]. Indirect interactions are represented by communication with tuple spaces [36] and blackboard architectures [99]. 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 [90, 91]. 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 motivations 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. Computational argumentation models is a notable example. Computational argumentation is much less structured than protocols, owing to the way humans argue in practice [39]. 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 [44, 15, 14], though often seen as the ‘absence of global control’ [112]. Practical examples of autonomous agents in the MAS community are auction agents that execute on behalf of their owners, following predefined strategies [13, 122, 134, 63].

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 [111]. 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" [137]. 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) [27, 29]. 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 [28]. 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:

<table>
<thead>
<tr>
<th>Definition: Heterogeneity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A MAS is heterogeneous when agents or the infrastructure of the system are developed by different means.</td>
</tr>
</tbody>
</table>

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 [18]. Tuple spaces, tuple centers, and more generally the notion of environment are some instances of such middleware in MAS [75, 22, 129].

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 management 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.

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 [35]. 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.
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 subsequent stages and refinements of the development process, down to the implementation in chapter 7, with particular focus on 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 [13, 122, 134, 93]. 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.

![Dependency network among actors on the market](image)

Figure 1.3: Dependency network among actors on the market

When agents need to trade items, they use the classical Contract Net protocol (CNet) [114] 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) [33]. 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 [124] and AUML [2], but it has been reduced to the minimum required to describe the case study.

**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 (timeout), 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 them3. 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

---

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.
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 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).
1.4 ORGANIZATION

Another type of exception that should be managed by agents is the case of the ‘agent death’ [64], 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 7, 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, with three distinct parts.

Agent Exceptions and Management by Autonomous Entities. The first part is gathered chapters relative to the study of the concept of agent exception in the case of autonomous agents. 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.
Robust Agent Execution Model and Engineering Framework. Part II is entitled aims at presenting our work on managing agent exceptions. Chapter 4 develops an agent execution model including our exception management approach, 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 7 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 experimental 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.

Uniφ: Building Agent Systems Robust to Exceptions. Part III proposes an integration of our agent execution model with the sentinel approach, as a mean to leverage both agent- and system-level exception management capabilities. Chapter 8 explains the relationships of the different levels and how they can complement effectively. We then propose the Uniφ framework for joint use of the two levels. Chapter 9 details how the framework is applied to the Energy consortium case study, along with a discussion on the perspectives of the approach and its complexity.

Finally, chapter 10 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.
Part I

Agent Exceptions and Management by Autonomous Entities
Related Work

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 aims at presenting 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 \([42, 40, 41]\). 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 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.

\(^1\)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 \([41]\). 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.
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.

```java
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 exist, 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.

![State-chart describing the operational semantics of exception handling in many programming languages](image)

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

The state-chart shows on the left-hand side (white part of the chart) the ex-
2.1. EXCEPTIONS IN PROGRAMMING LANGUAGES

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 [43], C#, the different versions of C++ [118], 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 [84].

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 [108, Chapter 19]. That is, conditions stand at a higher level than exceptions, so that any event can be handled the same way.

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 where 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 [121, 82]. 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 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 [82].
The guardian assists a client-server system shown in Fig. 2.4 that implements the
‘primary-backup’ approach to deal with server-side failures [120].

Clients connect to a server to get some services executed in a usual request/re-
ply fashion. Behind the scene on the server-side, the primary-backup is a replica-
cation 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
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

---

**Figure 2.3:** State-chart describing the operational semantics of exception handling
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 [135]. Fig. 2.6 helps to illustrate this concept.

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 [135])](image)

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 [31, 23].

**Cooperation exception handling.** Cooperation exception handling elaborates on a cooperation model among distributed objects involved in *multi-party interactions* [52]. 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 programming 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 [52])

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 ar-
guments 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 [64].

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 Banatre that introduces exception handling constructs and runtime support to an ADL [53]. 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 [119]. The issue with COTS in practice is the actual
2.3. ARCHITECTURE-LEVEL AND COMPONENT-LEVEL EXCEPTIONS

The 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 [20]. 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 [20, 61, 64]. 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 [135, 103]. 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' [103]. 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 [81, 66]. 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.

$$\forall X, \text{bird}(X) \land \neg \text{abnormal}(X) \supset \text{flies}(X)$$

(2.1)

The problem with this formula is that an inference engine cannot derive any conclusion from the only knowledge that ‘$X = \text{Tweety}$’ is a bird. The logical formula requires explicit knowledge relative to the abnormal predicate, either abnormal(Tweety) or $\neg \text{abnormal}(\text{Tweety})$ 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.

$$\forall X, \text{bird}(X) : \text{flies}(X) \supset \text{flies}(X)$$

**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. The detail and formal mechanism of circumscription is described in the original work [78, 79], and what matters for the engineering of agent exceptions is the informal semantics of the approach. The problematic formula becomes:

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

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

**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 $\text{bird}(X)$. 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 without any handler available nor propagation option (i.e. lack of knowledge to be compared to not knowing $\text{abnormal}(X)$). 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.

---

2Circ is a predicate over predicates, introducing a single second-order term (Circ itself) to achieve circumscription of a first-order theory.
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 [107, 106].

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 [107, 106]. 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.

![Hierarchy organization of a MAS](image)

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 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 [49], 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’ [98]). When an exception is detected in interactions or agent states, the sentinels execute specific code to recover a desired state.

A detailed application from Hagg 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 [61]. 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 [110]. This work focuses on analyzing the contents of FIPA-compliant agent communication languages [27]. 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’ [49], 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 [110]. 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 [64]. The sentinels function similarly to the original model of Hagg, 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.

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 [73, 72]. This approach relies on commitment protocols to specify how autonomous agents interact in an open system. Commitment
25. EXCEPTIONS IN MAS RESEARCH

Figure 2.11: Reliability database in the sentinel approach

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 foresee—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 [61].

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 [73]. 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 On-line execution monitoring

Monitoring the execution of agent systems at runtime has been proposed by Kaminka and subsequent work as a necessary agent activity to recover from exceptional situations [59, 60, 68, 48]. Execution monitoring relies on overhearing communications among agents in a given neighborhood, so that the additionally collected information can serve in analyzing the state of the system. Execution monitoring is in fact a mean to detect actively when the situation turns wrong for an agent, according to its knowledge such as a shared plan in the work of Kaminka. It provides consequently a number of observation strategies that allow agents to trigger exception management mechanisms. The focus of existing work is to elaborate observation strategies that allow detecting efficiently exceptional situations and providing management mechanisms with relevant contextual information.

Existing observation strategies rely on social or structural relationships of agents in the system, such as team patterns or social organizations. For instance, recent work from Gutnik focused on hierarchical organizations to determine selection criteria of agents to monitor in the hierarchy, with interesting results, notably the advantage to monitor most active agents in priority, the limited impact of the type of hierarchy (pyramid, inverse-pyramid, etc.), and the limited impact of the hierarchy depth [48].

Execution monitoring and the consequent issues of monitoring selectivity are in our sense essential techniques as for exception management in MAS, and in distributed systems in general. The cost incurred by active monitoring can be controlled by appropriate selection strategies, and the notable contribution to MAS is the capability to search for problems in the system, so as to actively search for a remedy. Execution monitoring contributes then to the aforementioned problem of awareness of the context in managing exceptions. These mechanisms are thought of as an important capability that agents can acquire in dealing with exceptions, in extension to the execution model introduced in this document.

2.5.5 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 [88]. 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 [89]. 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.

2.5.6 SaGE in the MadKit platform

Souchon et al. proposed the SaGE framework (acronym for Agent exception handling system) [115]. SaGE extends the exception handling system of Java with facilities to handle issues specific to autonomous agents in the MadKit platform [70]. 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 [25]) 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 issues 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.
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 [41, 42, 40].

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.
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.

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 [129].

### 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. 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-

1The 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 handling actions’ (or 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](image)

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’ [64]. 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.
3.3. EXCEPTION SPACE IN MULTI-AGENT SYSTEMS

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 handling action types that can be created.

<table>
<thead>
<tr>
<th>Agent Level</th>
<th>Known</th>
<th>Unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinated</td>
<td>ACK</td>
<td>ACU</td>
</tr>
<tr>
<td>Standalone</td>
<td>ASK</td>
<td>ASU</td>
</tr>
<tr>
<td>Code Level</td>
<td>CK</td>
<td>CU</td>
</tr>
</tbody>
</table>

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

**The knowledge dimension.** First of all, exceptions are usually either *known* or *unknown* 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 [116]), while complex reasoning agents would exploit the situation, such as KGP agents [58].

**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 interaction with other agents, the exception is classified as *standalone*. Corresponding

---

2Related work also exploit this distinction. They however use the terminology ‘expected’ and ‘unexpected’ [73]. Given that exceptions are usually considered as unexpected situations, this document exploits the ‘known’ quality instead.
actions for handling the standalone case do not need communication, so interaction protocols are not necessary. When the agent considers the handling requires to coordinate with other agents, the exception is then deemed as coordinated. Corresponding actions for handling then require structured coordination means, typically by communication according to interaction protocols. 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 [120, 61, 64, 82]. 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. 3.1. 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 [22, 126]. 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 control 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 [135, 82]. 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.
Part II

Robust Agent Execution Model and Engineering Framework
Agent exceptions are seemingly frequent events in the dynamic environments where agents can execute. Robustness to agent exceptions essentially aims at endowing the agents with appropriate capabilities to continue correct and useful executions despite the occurrence of such exceptional situations. The aim of this chapter is to present an agent execution model that encompasses exception management mechanisms for improving robustness. The model is detailed formally, so that a practical engineering framework can be derived from it in the next chapter to provide software designers with guidance and focus on (i) the services included in the execution model and (ii) the tasks to achieve in order to design a 'robust agent'.

This chapter is organized as follows. The next section presents the assumptions of this work about the target agent model and the scope of the agent capabilities. Section 4.2 and 4.3 present the syntax and semantics of the model respectively.

4.1 Assumptions and focus

Various kinds of agents were developed to address different research issues, ranging from architectures based on reasoning models [98, 58], to ant-like models [21, 11], to 'hybrid architecture' [37, 31, 32, 51, 55, 56]. The model presented hereafter differs from the existing approaches as it 'wraps' them with our proposal for exception management. In other words, the rationale of our model is to extend existing architectures with necessary mechanisms for managing exceptional situations. The model is responsible for detecting exceptions and presenting required information to the 'core' architecture that decides the reaction of the agents. We suppose this core architecture computes 'decisions' and 'plans', but these should be understood in a broad and flexible manner. For example, reactive ant-like agents do not plan per se, as defined in AI, but we assume that a plan can be a single action in such a case.

The model is built and presented under the following assumptions. The origin of these assumptions will be refined throughout the presentation of the model.
• Agents are owned by a human or another process, which are the only entities allowed to 'override' the agent autonomy for configuration and command sakes.

• Agents are goal-driven. We suppose that agents are given goals by their owners, and they are committed to fulfill these goals in their execution environment. An alternative approach to goal-driven agents is based on activity theory [16, 87]. The goal-driven view on agents is however more appropriate as for software engineering concerns in the present state of research, and the activity theory view is left for future work.

• Goals are supposed 'unit goals', i.e. goals are not decomposable into sub-goals. An agent then executes goals in series and does not need to reason on how to decompose goals. This simplification allows us to focus on the core problem: The mechanical reactions of the agent when exceptional conditions occur with a goal. The generalization is concretely a refinement of the proposed mechanisms, thus introducing no new essential technique.

• Agents achieve goals by building and executing plans. As stated earlier, plans can be reduced to their simplest forms (a single action), so that reactive agent architectures are also covered by our approach. The planning problem in those cases become an action selection problem, while it remains a general planning issue in the general case as defined in AI.

• We suppose that planning abilities encompass panning for joint activities, as in the work of Kaminka [59] or Grosz & Kraus [45].

• Plans are not de-feasible. In other words, we do not consider the case where the agent is able to 'undo' actions when time allows. When a plan fails, the agent tries to build a new plan, which can however contain actions that undo previous ones, so this apparent limitation is essentially technical.

• The agent can execute plans in parallel. In particular, the agent generates plans when it initiates a single or joint activity, and when it is invited to take part into an activity.

• Consequently, interaction protocols can be executed in parallel, in the frame of one plan, or in different ones.

• Actions of agents in the environment are either message exchanges (then part of interaction protocols) or operations effective on the environment (e.g. for situated agents or when invoking, say, a database).

• A shared ontology is assumed in the system and available to all agents, i.e. we assume no interoperability problems on the message semantics.
4.2 Syntax of the Agent Execution Model

The model captures the reactions of the system facing exceptional situations, with regards to the agent process owner (notably the human user). The model defines an explicit boundary between the agent capabilities and the owner responsibilities. Goals are assigned to the agent by the owner, and a failure to achieve a goal is an exceptional situation that must be managed by the owner, not the agent, to ensure the agent behaves safely (in the software safety sense).

Agents generate and execute plans toward a goal, and it is up to the agent to manage exceptions in the planning process by re-planning as necessary. Failure to plan the achievement of a goal links to the previous case. The agent executes protocols and operators in the environment along the execution of a plan. Failure to execute protocol and operator is managed by the agent, but we assume in the model that alternative executions are specified by the agent designer, depending on the execution context. The execution model then provides a mechanical mean to exploit adaptively the alternatives provided by the designer. AI techniques can clearly be integrated to elaborate on the automatic management of unknown exceptions in this framework, and we conducted work in such directions (e.g. unpublished studies based on Case-based reasoning and abductive reasoning to generate alternatives to exceptions in protocols), but the present work focuses on the mechanical framework that supports such techniques. The contribution of the model is therefore to provide a formal account of the above mechanisms, down to an engineering framework and a software architecture.

4.2 Syntax of the agent execution model

The syntax of the model relies on the first-order predicate logic to represent the knowledge and capabilities of the agent. We define three syntactic elements, namely literals, formulas, and predicates.

4.2.1 Literals and formulas

Literals Table 4.1 compiles the literals exploited in the model.

<table>
<thead>
<tr>
<th>Kind</th>
<th>Syntax</th>
<th>Kind</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Stamp</td>
<td>$t_i \in \mathcal{T}$</td>
<td>Domain Knowledge</td>
<td>$d_i \in \mathcal{D}$</td>
</tr>
<tr>
<td>Agent</td>
<td>$a_i \in \mathcal{A}$</td>
<td>Goal</td>
<td>$g_i, (g_i, t_i) \in \mathcal{G}$</td>
</tr>
<tr>
<td>Plan</td>
<td>$plan_i \in \mathcal{Plan}$</td>
<td>Protocol</td>
<td>$proto_i \in \mathcal{Proto}$</td>
</tr>
<tr>
<td>Operator</td>
<td>$op_i \in \mathcal{O}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Literals in the model

Time is explicitly represented with fresh time-stamp literals. The domain knowledge is represented as a set of facts. Domain knowledge complies implicitly with
an ontology, so that it also serves to express message contents as formulas (see next section). It also refers to communicative act names, such as the one defines by the FIPA [27] or KQML [67]. Agent literals refer to the agents of the system, with unique identifiers represented by the literals. Agents are assigned goals that we assumed indivisible in this model. Plan, protocol, and operator identifiers are used to represent instance of these elements.

**Formulas.** Our model defines two types of first-order predicate logic formulas.

1. **Message contents** describe the contents of communicative acts, similarly to the content languages like FIPA-SL [30], with a syntax limited to first-order predicate logic.

2. **Percepts** are conjunctive formulas $p_i \in \text{Percept}$ expressing a perceived state of the environment as a conjunction of facts.

### 4.2.2 Predicates

Protocols, actions, and plans executed by the agents are expressed as predicates, along with the set of application-dependent operators like $op$, and the specific $transfer$ for message exchanges among agents. Operators vary depending on the application requirements, so that $op$ can represent the functions of an connection API to a database or a web-service, or the interface for primitive commands of a robot piloted by a software agent.

#### 4.2.2.1 Operators and message transfers

**Operators.** When agents act in the environment except communication, they apply application-dependent operators. The set of operators is related to the capabilities of the agent, and the following predicate represents an abstraction of such an operator $op$ in our model.

$$op(arg[n])$$

An operator is a predicate over a list of arguments, represented as the $arg[n]$ $n$-tuple.

**Message transfers** Message transfers are primitive interactions when agents exchange message to communicate. $transfer$ is a specific case of operator, where
the list of arguments is a 7-tuple

\[ \text{transfer}(\text{conversation} - id, \text{protocol}, \text{from}, \text{to}[n], \text{perform}, \text{message}) \]

Where:
- \( \text{conversation} - id \in \mathcal{D} \)
- \( \text{protocol} \in \mathcal{Proto} \)
- \( \text{from} \in \mathcal{A} \)
- \( \text{to}[n] \in \mathcal{A}^n \)
- \( \text{perform} \in \mathcal{D} \)
- \( \text{message} \) is a formula.

A \textit{transfer} is similar to the form of FIPA-ACL messages [27], with specific focus on the necessary information for our execution model (the fields of a \textit{transfer} message are a sub-set of the fields of a FIPA-ACL one). This format of message can therefore be conveniently replaced by standard ones, if necessary.

### 4.2.2.2 Protocols and protocol enactment

Protocol names in \( \mathcal{Proto} \) refer to tree structures that specify series of message transfers in a given rationale. For example, the version of Contract net protocol exploited in the case study is represented as a tree of message transfers between client and participants (see Fig. 1.4, page 13). The syntax is as follows.

\[ \text{protocol} \text{\_\_\_\_\_\_type}(\text{name}, \text{role}[n], \text{tree}) \]

Where:
- \( \text{name} \in \mathcal{D} \) (e.g. Contract net, circular voting, request)
- \( \text{role}[n] \in \mathcal{A}^n \)
- \( \text{tree} := \quad \text{kill} \quad | \quad \text{Terminate dependent protocol} \)
  \[ \text{transfer}(\text{arg}[]) \quad | \quad \text{Transfer a message} \]
  \[ \text{tree}, \text{tree} \quad | \quad \text{Succession} \]
  \[ \text{tree} \cup \text{tree} \quad | \quad \text{Choice} \]

The above definition of protocols specifies a name, a list of \( n \) roles that participate\(^1\), and the tree structure of the protocol. In the case of the Contract net protocol between a client and a participant, we have the following syntax (only the

---

\(^1\)The multiplicity of roles is represented by \( n \), i.e. two instance of a role, e.g. participants, shall be explicitly noted with fresh and independent literals, such as \text{participant}_1 \) and \text{participant}_2.
performative field is represented to avoid clutter):

\[ \text{protocol}_\text{type}(\text{cnet}, [\text{client}, \text{participant}], \text{tree}) \]

Where:

\[ \text{tree} ::= \text{transfer}([\text{cfp}, \text{offer}]), \]
\[ \text{transfer}([\text{refuse}]) \]
\[ \cup \]
\[ (\text{transfer}([\text{propose}]), \text{transfer}([\text{reject}])) \]
\[ \cup \]
\[ (\text{transfer}([\text{propose}]), \text{transfer}([\text{accept}]), \text{transfer}([\text{failure}])) \]
\[ \cup \]
\[ (\text{transfer}([\text{propose}]), \text{transfer}([\text{accept}]), \text{transfer}([\text{result}])) \]

Protocols are static data structures publicly available to agents. When agents are involved in a protocol, they maintain a protocol enactment knowledge structure [9], defined as the instantiation of a protocol. Protocol enactments introduce additional notations relative to maintaining knowledge of the execution state, i.e. knowledge of the possible actions and active commitments for each agent.

\[ \text{protocol}(\text{identifier}, \text{protocol}_\text{type}, \text{agent}[n], \text{history}, \text{state}, \text{dependency}) \]

Where:

\[ \text{identifier} \in \text{Proto} \]
\[ \text{protocol}_\text{type} \in \mathcal{D} \]
\[ \text{agent}[n] \in \mathcal{D}^n \]
\[ \text{history} \text{ contains the series of transfers along the enactment.} \]
\[ \text{state} \text{ is either running or suspended.} \]
\[ \text{dependency} \text{ is the identifier of an action running instead of this protocol.} \]

\text{Identifier} identifies the protocol instance uniquely in the system. The \text{type} specifies the kind of protocol that is instantiated, and should therefore be known by the agent. The \text{agent list} is the assignment of role to agents, and it corresponds to the role list of the protocol type. The \text{history} compiles the series of message exchanges that occurred in the enactment of the protocol, so that the agent can keep track of the event. In the exception model, the last event of the history is exploited. The \text{state} of the protocol labels the protocol enactment as either \text{running}, when the agent is executing the protocol (processing or waiting for messages), or \text{suspended}, when the protocol is interrupted, typically in case of exceptional conditions. A \text{dependency} of protocol \text{p} is an action instance identifier \text{a} that states \text{p} is suspended and waits for \text{a} to complete (see next section for action definition). The rationale of the dependency is to describe situations where the protocol is suspended in case of exception and waits for another action to handle the situation. Such action is either an operator or another protocol and we call it the ‘exception handler’.
4.2. SYNTAX OF THE AGENT EXECUTION MODEL

The exception mechanism exploits both the dependency and the state information, although the dependency implies the state of the protocol. The state information is thus redundant, but it is maintained explicit for the readability of the formal model.

4.2.2.3 Action

Action predicate. Actions are of two possible types, namely the application of an operator or the execution of a protocol. An action can have both an action context and a rational effect.

\[
\text{action} ::= \text{op}(\text{arg}) | \text{protocol}(\text{arg})
\]

Apply an operator

Execute a protocol

Activation context and rational effect predicates. Actions can be executed in different contexts, depending on the application domain and the strategy of the agent decision process. In particular, an action can be used in the context of the main activity of the agent, and it can also be used in the context of management of an exceptional situation. The activation context allows to specify how to relate an event to some reaction capabilities of the agent (actions) in a contextual, thus adaptive fashion. The action activation context is noted as follows. It is a predicate that associates an action name to a conjunction of literals.

\[
\text{context}(<\text{action}, \text{expression}>)
\]

Actions effect in the environment and agents can draw some expectations on the effects they are waiting for. The rational effect serves to specify what an action should produce for an agent in a given activation context. The effect is also a conjunction of literals.

\[
\text{reffect}(<\text{action}, \text{expression}, \text{effect}>)
\]

4.2.2.4 Plans

Plans are represented as predicates, referring to tree structures.

\[
\text{plan}(<\text{identifier}, \text{agent}, \text{tree}, \text{history}>)
\]

Where:

\[
\begin{align*}
\text{identifier} & \in \text{Plan} \\
\text{agent} & \in D_n \\
\text{history} & \text{is the current series of actions executed in the plan.}
\end{align*}
\]

\[
\text{action}(\text{arg}) \otimes (c, r) |
\]

Action in the system with action context and rational effect

\[
\begin{align*}
\text{tree} ::= & \text{tree}, \text{tree} & \text{Succession} \\
& \text{tree} \cup \text{tree} & \text{Choice} \\
& \text{tree} \parallel \text{tree} & \text{Parallel}
\end{align*}
\]
The identifier and history have the same purpose as in protocols. The agent list corresponds to agents involved in this possibly shared plan. The tree structure refers to actions. Actions are executed in a certain activation context $c$ and are expected to yield a rational effect $r$, depending on the plan design and the domain application. In other word, context and effects can be specified by the designer. Consequently, the same action may be executed in various situations depending on the contextual constraints. The plan tree structure contains a parallel operator that allows the execution of several protocols simultaneously.

### 4.3 Semantics of the agent execution model

The purpose of this section is to present an operational semantics of the agent execution model, including our agent exception management model. The semantics is presented as a state rewriting procedure integrated into algorithms. We first present convenience operators, the definition of the state of an agent, and then the complete procedure over the agent state.

#### 4.3.1 Preliminary semantics: Plan and protocol operations

The next operation. The next operation is a convenience that formally specifies the next possible actions in plan or next message transfers in protocols, depending on their execution state.

\[
\text{next} : \text{Plan} \cup \text{Protocol} \rightarrow \mathcal{P}(\mathcal{O} \cup \text{Protocol}) \\
\text{plan}_i \mapsto \{\text{op} \in \mathcal{O} \setminus \{\text{transfer}\} \mid \text{plan}_i.\text{history} \cup \{\text{op}\} \text{ sub-tree of plan}_i.\text{tree}\} \cup \\
\{\text{proto} \in \text{Proto} \mid \text{plan}_i.\text{history} \cup \{\text{proto}\} \text{ sub-tree of plan}_i.\text{tree}\} \\
\text{proto}_i \mapsto \{\text{transfer} \mid \text{proto}_i.\text{history} \cup \{\text{transfer}\} \text{ sub-tree of proto}_i.\text{tree}\}
\]

In other words, the ‘next’ operation determines the set of operators and protocols that were planned after the current action of a plan. It similarly determines the set of message transfers that are specified after the transfer in a protocol.

#### 4.3.2 Agent state

An agent state is defined as a tuple, including an explicit representation of agent expectations.

\[
\text{Definition: Agent state} \\
S := (\mathcal{G}, \mathcal{O}, \mathcal{Proto}, \mathcal{Plan}, \mathcal{D}, \mathcal{O}^*, \mathcal{Proto}^*, \mathcal{X})
\]
The different sets of the agent state refer to variables necessary to describe the runtime, either in terms of literals, formulas, or predicates.

\[ G \] is the set of goals.
\[ O \] is the set of known operators.
\[ \text{Proto} \] is the set of known protocols.
\[ \text{Plan} \] is the set of plans.
\[ D \] is the domain knowledge.
\[ O^* \] is the set of operators under execution.
\[ \text{Proto}^* \] is the set of protocol enactments.
\[ X \] is the set of expectations.

Plan and expectation sets are further refined as follows. \( \text{Plan} = \text{Plan}_{\text{run}} \cup \text{Plan}_{\text{fail}} \) distinguishes plans that are run by the agent, from past plans that failed to complete. When there is no ambiguity on the use, we note \( \text{Plan} \) for \( \text{Plan}_{\text{run}} \).

The expectation set is split into sub-sets to distinguish the different formats of expectation elements. \( X = X_G \cup X_{\text{Plan}} \cup X_{\text{Proto}} \cup X_O \). The detail of each expectation format is described within the operational semantics. All expectation elements follow the pattern \((x, t)\), where \(x\) relates to either a goal, plan, protocol or operator, and \(t\) refers to an expected time.

We also define a view as a 'cross-cutting' notion. A view is a sub-set of the agent state that focuses on a particular interest.

**Definition:** View

A view is a tuple that represents partially the state of an agent. E.g.:

- \( V := \{G\} \) is the view on the agent goals
- \( V := S \) is the complete view on the agent state
- View of a goal: Let \((g, t) \in G\) for some time \(t\). The view \( V_g := \langle \text{Plan}_g, O_g^*, \text{Proto}_g^*, D_g, X_g \rangle \) is the view of the agent state restricted to all state information related to goal \(g\).

Views serve in the algorithms to set forth the salient agent knowledge during the execution process, and to formally avoid border-effects of under-specified models.

Similarly to the view of a goal, we will use the view of a plan and a protocol on the agent state. When these notations are necessary in the model (especially for the semantics), they will be defined at time of use.

### 4.3.3 Execution procedures

#### 4.3.3.1 Overview of the procedures

The procedures describe the execution of agents with regards to their dynamic knowledge, i.e. goals, plans, protocol enactments, and operator execution. The
static knowledge represented by protocols, operators, and domain knowledge is exploited by the procedure, but it is not modified. In other words, the agents are not supposed to learn in this model, and learning capabilities can be added as independent extension.

The execution procedures are part of our agent framework, i.e. the system designer does not need to write the corresponding mechanisms, which are defined and ready-to-use. The designer shall however provide a number of application-dependent features, as we will detail in section 6. In particular, the designer shall choose (joint) planning and decision modules, and our framework wraps them in dealing with exceptional situations. In addition, we suppose available to the agent a reporting system, such as a front-end user interface. The three supposed functionalities are represented by the abstract functions \( \text{plan}(\text{arg}) \), \( \text{decide}(\text{arg}) \), and \( \text{report}(\text{arg}) \), with trivial signatures.

The overall procedure is decomposed into sub-procedures depending on the execution context and their impacts on the agent knowledge. The critical matter for exception management is to determine the context of an exception occurrence, so that it can be treated appropriately [40]. Fig. 4.1 gives an informal account of the context identified in the agent execution model, and their nested structure.

![Figure 4.1: Nested structure of the agent execution contexts](image)

We define four execution contexts in our approach. The agent global execution context is for reasoning about goals, as we supposed goal-driven agent models. In the execution of a goal, an agent has to execute plans, so the execution of goals contains contexts of plan executions. Each plan is a tree of operator and protocol execution, so that the plan execution context contains contexts of both operator and protocol executions.

The highest-level procedure corresponds to the goal execution context. For each nested context corresponds a sub-procedure, with a restricted view on the agent state depending on the context ‘depth’ in the agent execution process. Table 4.2 introduces the name of the sub-procedures depending on their scope, along with the corresponding views. An extra timeout sub-procedure is introduced to facilitate
the management of time events. It could have been integrated equivalently in the other sub-procedures.

<table>
<thead>
<tr>
<th>Level</th>
<th>Procedure</th>
<th>Procedure View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>exec(S, t)</td>
<td>V_G = (G, Plan, D, X)</td>
</tr>
<tr>
<td>Plan</td>
<td>exec_plan(S, t)</td>
<td>V_Plan = (O, Proto, Plan, D, X, O*, Proto*)</td>
</tr>
<tr>
<td>Protocol</td>
<td>exec_proto(S, t)</td>
<td>V_Protol = (D, X, Proto*)</td>
</tr>
<tr>
<td>Operator</td>
<td>exec_op(S, t)</td>
<td>V_O = (D, X, O*)</td>
</tr>
<tr>
<td>Timing</td>
<td>timeout(S, t)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Reference table of the execution procedure

The goal level can react directly to commands received by the agent process owner. These commands essentially allow the human user to delegate and drop goal assignments to the agent. The plan level does not react directly to environmental triggers. Plans are generated according to goals and evolve according to protocol and operator executions, which are all internal knowledge and structures of the agent. Protocol and operator react to message transfers (receptions) and operators (feedback result) respectively.

In addition to these reactions to environmental triggers, Fig. 4.2 shows the relationship between the execution at different levels.

![Figure 4.2: Relationship between the different operators depending on their level in the agent architecture. Arrows are ‘calls’ to the target procedure](image-url)
Although agents can be deliberative or else in this approach, there must be a trigger at some point of the agent execution to 'bootstrap' the deliberation process. On the figure, such triggers can be either a command from the owner, or the reception of an event from the environment, either a message transfer or the feedbacks from the application of an operator. The figure then illustrates the control flow in the agent architecture.

- Commands cause a descending flow in the execution: Addition or removal of a goal implies the creation of plans and then the execution of protocols and operators.

- Message transfers and percepts cause a rising flow in the execution. Their occurrences at the protocol and operator levels imply upward modification of plan states and the revision of goals.

The exception mechanism is at the plan level, as exception management suggest a 'meta-level' relative to protocol and operator executions. As protocol and operator are executed in the context of a plan, the plan level is the appropriate one to integrate the exception management functionalities.

An additional exception management could be distinguished at the goal level, regarding the management of plan exceptions. In fact, the corresponding mechanism is integrated in the eval procedure at the goal level, as it is not significantly remarkable to be separated.

The participation procedure is a specific feature at the plan level that manages invitations to participate in a protocol from other agents. This specific approach aims at simplifying the presentation of the different cases, similarly to the distinction of the timeout functionalities (not represented on the figure).

### 4.3.3.2 Goal procedure

The procedure that describes the agent reasoning on goals is expressed in the following definition box. The procedure relies on four operations that are response to distinct commands.

**Definition:** Goal procedure over state $S$ at time $t$

$\text{exec}(S, t) ::= (\text{new \_goal}(S, t), \text{drop \_goal}(S, _),$ 
$\text{run}(S, t), \text{evaluate}(S, \text{plan}, t))$

The goal procedure is executed on an event base and accepts three triggers.

- New goal command from the agent owner that entails $\text{new \_goal}$
- Goal removal command from the agent owner that entails $\text{drop \_goal}$
4.3. SEMANTICS OF THE AGENT EXECUTION MODEL

- Evaluation of plan state, on demand by plans

The goal procedure relies on the corresponding view of the goal on the agent state, as defined on page 61. The operational semantics of the goal procedure is formally defined in the following four algorithms. First, new_goal merely inserts an incoming goal from the owner and calls the run procedure to initiate the agent activity.

```
Input: Agent state S, Goal g, Time t
G ← G ∪ \{(g, t)\};
run(S, t)
```

Algorithm 4.1: new_goal(S, g, t)

The drop_goal procedure removes a goal from the agent goal list and cleans up the knowledge base of the agent, so that the agent does not execute any more expression related to the dropped goal.

```
Input: Agent state S, Goal g
G ← G \{g | (g, t) ∈ G};
\{ Clean-up the corresponding goal view \};
X_g ← X_g \{plan, t | plan, t ∈ V_g\};
Plan ← Plan \{plan | plan ∈ V_g\};
Proto∗ ← Proto∗ \{proto | proto ∈ V_g\};
O∗ ← O∗ \{op | op ∈ V_g\};
```

Algorithm 4.2: drop_goal(S, g)

The run procedure calls the planning capabilities of the agent to attempt creating a plan toward assigned goals. Planning failure is seen in this model as an agent exception, but the management is left to the owner, as our purpose is to elaborate an engineering framework, which must define a reasonable boundary on the agent capabilities. As goals are the rationale of the agent execution, we consider that management of goal completion failure is out of the agent boundary. This type of functionalities can be the subject of further investigation. The present model requires the agent to report the problem to the owner and proceed with the next goal, if any.

The evaluate procedure reacts to calls from the plan level to update the agent knowledge relative to goals. The evaluation reports goal completions to the owner and loops the agent execution to proceed with the next goal, if any.

4.3.3.3 Plan procedure

The plan procedure describes the execution of plans and the management of interaction protocols and operators necessary to achieve the plan. The definition of the plan procedure consists of four sub-procedures.
**Algorithm 4.3: run(S, t)**

**Input:** Agent state S, Time t

Let g so that \((g, t) \in G\);

\{ Build a fresh plan in \(\mathcal{Plan} \setminus \mathcal{Plan}_{\text{fail}}\) to achieve g \};

\(\text{plan} \leftarrow \text{plan}(g, S)\);

if plan exists then

\{ Expectation: Termination of the plan by time \(t\) \};

\(X_G \leftarrow X_G \cup \{(\text{plan}, t)\}\);

\{ Execution of the plan \};

\(\mathcal{Plan} \leftarrow \mathcal{Plan} \cup \{\text{plan}\}\);

\(\text{run}\_\text{plan}(S, t)\);

else

\{ Limit of the agent responsibility: Report and Loop \};

\(\text{report}(g, \text{fail})\);

\(\text{run}(S, t)\);

end

---

**Algorithm 4.4: evaluate(S, plan, t)**

**Input:** Agent state S, Plan plan, Time t

\{ Clean-up the knowledge base \};

\(X_G \leftarrow X_G \setminus \{(\text{plan}, t)\}\);

\(\mathcal{Plan} \leftarrow \mathcal{Plan} \setminus \{\text{plan}\}\);

\{ Result analysis \};

Let g \(\in G\), so that plan \(\in \mathcal{V}_g\);

if \(\exists\) formula \(f \in D, g \vdash f\) then

\{ Goal in domain knowledge, as effect of the plan \};

\(\text{report}(g, \text{success})\);

else

\{ Simple instance of exception management: Re-plan \};

\(\mathcal{Plan}_{\text{fail}} \leftarrow \mathcal{Plan}_{\text{fail}} \cup \{\text{plan}\}\);

\(\text{run}(S, g)\);

end

---

**Definition:** Plan procedure over state S at time t

\[ \text{exec}_\text{plan}(S, t) := (\text{run}\_\text{plan}(S, t), \text{evaluate}_\text{plan}(S, \text{action}, t), \text{exception}_\text{plan}(S, \text{action}, t), \text{participation}(S, \text{transfer}, t)) \]

The \(\text{run}\_\text{plan}\) procedure is invoked by the goal level in order to execute a plan. It essentially selects the next action specified by the plan and initiates
4.3. SEMANTICS OF THE AGENT EXECUTION MODEL

its execution, depending on whether it is an operator or a plan. The exception management appears in this procedure with the management of expectations. The execution stores the expectation that the plan will complete by the specified time $t$.

**Algorithm 4.5: run_plan($S, t$)**

Let $plan \in Plan$;

$candidates \leftarrow next(plan)$;

if $candidates \neq \emptyset$ then

\{ Decide next action from the activation context of $S$ that will finish at $t_{act} < t \}$;

$(action[arg[]]) @ (c, r, t_{act}) \leftarrow decide(candidates, S, t)$;

if $action$ is defined then

\{ Expectation: Rational effect of the action by $t_{act}$\}

$X_{Plan} \leftarrow X_{Plan} \cup \{(action, r, t_{act})\}$;

$plan.history \leftarrow plan.history \cup \{action\}$;

\{ Execution of the action \};

if $action \in O$ then

$O^* \leftarrow O^* \cup \{action\}$;

run_op($S$);

else

$Proto^* \leftarrow Proto^* \cup \{action\}$;

run_proto($S$);

end

else \{ No action defined in the current execution context: Failure \};

evaluate($S, plan, t_{now}$);

end

else \{ Fail to run the plan on time, inform the goal level \};

evaluate($S, plan, t_{now}$);

end

The evaluate_plan procedure serves to react when actions are completed. It updates the agent knowledge about the executing plans and propagates to the goal level the results. In case a plan is still executing, the procedure loops on run_plan to process the next action. The procedure also aims at cleaning up the agent knowledge about past expectations about the plan and its state.

Even though the above algorithm does not mention about exceptional cases, the execution can proceed smoothly in any case. In fact, as the exception management phase replaces an action by another action with the same $(c, r)$ signature, the above algorithm sees no difference and proceeds the execution.
The plan level features the exception plan procedure to manage exceptions detected during operator and protocol executions. The procedure introduces handlers as an action that executes in place of the interrupted one, with the different cases of AxK and AxU as introduced in the classification of Tab. 3.1, page 46. The action selection process is then based on the activation context and rational effect approach. An eligible action for handling (thus becoming a handler) is selected if it is executable in the current context (satisfaction of the activation context) and if it produces the same rational effect than the interrupted action. This flexible selection mechanism yields an adaptive behavior of the agent.

In case the action selection returns no candidate action, the agent has no related capabilities to deal with the exceptional situation, which causes the failure of the related plan. In this approach, the agent does not manage the case AxU, i.e. when handling actions are unknown to the agent (see 3.1, page 46). Section 5 elaborates on some extensions of the model to introduce additional mechanisms and deal with a potentially wider range of cases.

The participation procedure specifies the reaction of the agent to invitations in joining interaction protocols. The rationale of the agent is to accept such invitation if (i) it knows the protocol and (ii) the rational effect of the participation contributes to one of its goals. Other acceptance rules could be defined, but this one presents the advantage to be economical and compatible with the agent’s owner commands.
4.3. SEMANTICS OF THE AGENT EXECUTION MODEL

Input: Agent state $S$, Action $action$, Time $t$

\{ Search for an alternative action that yields the same rational effect in the current context \};

$candidates \leftarrow \{ a \mid (\text{action}\cdot r \vdash a\cdot r) \land (a\cdot c \vdash \mathcal{D})\};$

if $candidates \neq \emptyset$ then

\{ Case of known handler case (AxK) \};

Let $handler \in candidates$;

if $handler$ is an operator then

\{ Case of ASK \};

$O^* \leftarrow O^* \cup \{ handler\};$

run$_{op}(S, t)$;

else

\{ handler is a protocol: Case of ASK or ACK \};

$Proto^* \leftarrow Proto^* \cup \{ handler\};$

run$_{proto}(S)$;

end

\{ Create a dependency to recover the execution in case of protocol interruption \};

if $action$ is a protocol then

$action\cdot dependency \leftarrow handler;$

end

else

\{ Cannot manage the exception (AxU): Failure of the plan \};

if $action$ is an operator then

$O^* \leftarrow O^* \setminus \{ action\};$

else

\{ action is a protocol \};

$Proto^* \leftarrow Proto^* \setminus \{ action\};$

end

evaluate$(S, plan, t now)$;

end

Algorithm 4.7: $\text{exception\_plan}(S, action, t)$

4.3.3.4 Protocol enactment procedure

Interaction protocols are the first case of execution of the agent in the environment. The procedure specifies how agents enact protocols and detect exceptional situations, based on recorded expectations.

Definition: Protocol procedure

$exec\_proto(S, t) ::= (\text{run\_proto}(S, t), \text{evaluate\_proto}(S, action, t))$
Input: Agent state $S$, Message transfer $transfer$, Time $t$

\{ Test on applicable protocols: 0 (ignore) or 1 (analyze) \}

$proto \leftarrow \{ p \in \mathcal{Proto} \mid (p = transfer.protocol) \land (transfer \in next(p)) \}$

if $proto \neq \emptyset$ then

\{ Decide whether participating is relevant or ignore otherwise \}

$outcomes \leftarrow \{ r \mid \text{reflect}(proto, – r) \in \mathcal{D} \}$

if $\exists (x, _) \in \mathcal{X}$ and $a \in outcome, x \vdash outcome$ then

Let $g$, so that $\exists (x, _) \in \mathcal{X}$ and $a \in outcome, x \vdash outcome$;

\{ Initiate the protocol in a new plan \}

$\mathcal{G} \leftarrow \mathcal{G} \cup \{ (g, t) \}$

$plan \leftarrow proto(arg[]) \otimes (\mathcal{D}, g)$

$proto\_history \leftarrow proto\_history \cup \{ transfer \}$

\{ Expectation: Termination of the plan by time $t$ \}

$\mathcal{X}_G \leftarrow \mathcal{X}_G \cup \{ \text{plan}, t \}$

\{ Execution of the plan \}

$\mathcal{Plan} \leftarrow \mathcal{Plan} \cup \{ \text{plan} \}$

run_plan($S, t$)

end

end

Algorithm 4.8: $\text{participation}(S, transfer, t)$

The $\text{run}_\text{proto}$ is triggered in the context of a plan. It executes the first message transfer of the plan to initiate the protocol. It also stores the next message transfers specified by the protocol as expectations (using the $\text{next}$ operation). In other words, the agent expects the next message received about the current protocol belongs to the set returned by $\text{next}$. Any other message will allow to detect an exceptional situation.

The $\text{evaluate}_\text{proto}$ procedure reacts to message receptions. Messages are matched with the current expectations, so that exceptional situations relative to the protocol can be detected. If the message was expected, the expectations are cleaned and the protocol is executed forward. Otherwise, the $\text{evaluate}_\text{proto}$ procedure further distinguishes messages that announce a new protocol and request for a participation of the agent, thus triggering the $\text{participation}$ procedure of the plan level. Other unexpected messages trigger the $\text{exception}_\text{plan}$ procedure.

4.3.3.5 Operator procedure

The semantics for operators is similar to the one of protocols. The essential differences are that operators are single actions (whereas protocols are series of operations), and stateless. In other words, operators are applied by the agent in the environment, which provides positive or negative feedbacks in return.
4.3. SEMANTICS OF THE AGENT EXECUTION MODEL

Input: Agent state $S$
Let $proto \in \text{Proto}^*$ with $proto\text{.state} = \text{running}$;
  { Decide next transfer };
    candidates ← next(proto);
  if candidates $\neq \emptyset$ then
    (transfer$[\text{arg}][\text{c}],[r],t_{out}) \leftarrow \text{decide}(\text{candidates},S);
      { Update the protocol history };
          proto\text{.history} ← proto\text{.history} \cup \{\text{transfer}\};
      { Expectation: Any of the next message following transfers in proto, with timeout $t_{out}$};
          forall message $\in$ next$(\text{proto})$ do
            \text{$X_{Proto} \leftarrow X_{Proto} \cup \{(message,t_{out})\};$}
          end
    if transfer = $\text{kill}$ then
      { Protocol subsuming another to terminate (handling) };
        \text{Proto}^* ← \text{Proto}^* \setminus \{p \in \text{Proto}^* \mid p\text{.dependency} = \text{proto}\};
    else
      { Execution of the transfer };
        send(transfer[\text{arg}][\text{c}][\text{r}]);
    end
  else
    { Protocol completed: Update and inform the plan };
      \text{Proto}^* ← \text{Proto}^* \setminus \{\text{proto}\};
      \text{evaluate\_plan}(S,\text{proto},t_{now});
  end

Algorithm 4.9: $\text{run}_{\text{proto}}(S)$

Definition: Operator procedure

$\text{exec\_op}(S,t) := (\text{run\_op}(S,t),\text{evaluate\_op}(S,\text{action},t))$

The $\text{run\_op}$ operator applies the operators in the environment. It stores additionally the rational effect of the operator as expectation.

The $\text{evaluate\_op}$ reacts to the feedbacks from the environment. It essentially cleans up the agent knowledge, informs the plan level about the results (which can impact several operators simultaneously), and detects exceptional situations when expectations are not met.

4.3.3.6 Timeout algorithm

The following $\text{timeout}$ algorithm is introduced to distinguish the reactive nature of the agent to the time aspect. It can be integrated to other algorithms without changing the semantics, but it is presented separately for readability concern.
Input: Agent state $S$, Message $transfer$, Time $t$

if $\exists t \in T, (transfer, t) \in X_{Proto}$ then
    { Expected transfer: Update knowledge and run the protocol forward }
    $D \leftarrow D \cup \{(\text{transfer from, transfer perform, transfer message})\}$
    $X_{Proto} \leftarrow X_{Proto} \setminus \{(m, \perp) | m.\text{protocol} = transfer.\text{protocol}\}$
    run.proto($S$, transfer);
else if $\{p \in Proto | transfer \in \text{next}(p)\} \neq \emptyset$ then
    { Unexpected transfer corresponding to a third-party protocol }
    participation($S$, transfer, $t$);
else
    { Unexpected transfer: Protocol-level agent exception, management in the plan. }
    $X_{Proto} \leftarrow X_{Proto} \setminus \{(m, \perp) | m.\text{protocol} = transfer.\text{protocol}\}$
    proto $\leftarrow \{p \in Proto^* | p = transfer.\text{protocol}\}$
    proto.state $\leftarrow$ suspended;
    exception.plan($S$, proto, $t_{now}$);
end

Algorithm 4.10: evaluate.proto($S$, transfer, $t$)

Input: Agent state $S$, Time $t$
Let $op[\arg[]] \otimes (c, r) \in O^*$
if $op$ exists then
    { Expect the rational effect and execute the operation }
    $X_O \leftarrow X_O \cup \{(op \, r, t)\}$
    apply($op[\arg[]]$);
end

Algorithm 4.11: run.op($S$, $t$)

The algorithm specifies the reaction of the agent to timeouts, as stored in the agent expectation set. Depending on the type of knowledge impacted by a timeout detection, the corresponding parts of the agent state are updated. In the case of protocol and operator, the procedure leads to exception.plan, whereas the timeout of a plan announces the failure of achieving the corresponding goal.
4.3. SEMANTICS OF THE AGENT EXECUTION MODEL

Algorithm 4.12: \texttt{evaluate}_{\text{op}}(S, percept)

\textbf{Input}: Agent state $S$, Percept $percept$, Time $t$

\begin{align*}
impact\_set & \leftarrow \{ \text{op}[\arg[\text{arg}]] \otimes (c, r) \in O^* \mid \text{op}.r \vdash percept \}; \\
\text{if } impact\_set \neq \emptyset \text{ then} & \\
\{ \text{Information of the percept is meaningful as for operator effects.} \}; \\
\text{forall } \text{op} \in impact\_set \text{ do} & \\
\text{if } \text{op}.r \in \mathcal{X}_O \text{ then} & \\
D & \leftarrow D \cup \{ \text{op}.r \}; \\
\mathcal{X}_O & \leftarrow \mathcal{X}_O \setminus \{ \text{op}.r \}; \\
O^* & \leftarrow O^* \setminus \{ \text{op} \}; \\
\text{evaluate}_\text{plan}(S, \text{op}, t_{now}); & \\
\text{else} & \\
\{ \text{Unexpected result: Operator-level agent exception, management in the plan.} \}; \\
\mathcal{X}_O & \leftarrow \mathcal{X}_O \setminus \{ \text{op}.r \}; \\
\text{exception}_\text{plan}(S, \text{op}, t_{now}); & \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\end{align*}

Algorithm 4.13: \texttt{timeout}(S, t)

\textbf{Input}: Agent state $S$, Time $t$

\begin{align*}
impact\_set & \leftarrow \{(x, t_{out}) \in \mathcal{X} \mid t_{out} \leq t\}; \\
\text{if } impact\_set \neq \emptyset \text{ then} & \\
\{ \text{Timeout for some plans, protocols, and operators.} \}; \\
\text{forall } x \in impact\_set \text{ do} & \\
\text{if } x \text{ is an operator then} & \\
\mathcal{X}_O & \leftarrow \mathcal{X}_O \setminus \{x.r\}; \\
O^* & \leftarrow O^* \setminus \{x\}; \\
\text{exception}_\text{plan}(S, x, t_{now}); & \\
\text{else if } x \text{ is a protocol then} & \\
\mathcal{X}_\text{Proto} & \leftarrow \mathcal{X}_\text{Proto} \setminus \{(m, \bot) \mid m.\text{protocol} = x\}; \\
\text{Proto}^* & \leftarrow \text{Proto}^* \setminus \{x\}; \\
\text{exception}_\text{plan}(S, x, t_{now}); & \\
\text{else if } x \text{ is a plan then} & \\
\mathcal{X}_G & \leftarrow \mathcal{X}_G \setminus \{(x, t)\}; \\
\text{Plan} & \leftarrow \text{Plan} \setminus \{x\}; \\
\text{Proto}^* & \leftarrow \text{Proto}^* \setminus \{\text{proto} \mid \text{proto} \in \mathcal{V}_x\}; \\
O^* & \leftarrow O^* \setminus \{\text{op} \mid \text{op} \in \mathcal{V}_x\}; \\
\text{run}(S, g); & \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\text{end} & \\
\end{align*}
Five

Extensions for more robust agents

The aim of this chapter is to introduce extensions of the execution model of agent that feature additional mechanisms to cope with some limiting assumptions of the main model. This chapter focuses on describing a broader execution model that addresses two issues.

Acquisition of knowledge about actions. Agents encompass an action selection method that may fail to find appropriate capabilities in the agent knowledge. The acquisition of additional actions from external resources is a straightforward extension of the agent execution model to collect useful actions that could be applied when necessary. This extension is inspired by the work from Klein and Dellarocas on their agent system infrastructure based on a handler repository [61].

Elaborated evaluation of handling actions. The previous capabilities introduces a problem relative to the agent autonomy. Acquisition of code from external sources may be hazardous, since the agent cannot necessarily trust this source. It is essential then that the agent evaluates the impact of acquired actions to determine whether they serve the agent purpose. The idea is for the agent to ‘simulate’ internally the rational effect of acquired actions, and then to decide whether they can be applied for its real activity. The evaluation of code is usually a computationally hard problem (otherwise, the termination problem would be feasible), but we present in this chapter a simple formal approach that can serve as the basis of more complex mechanisms.

We will also briefly discuss a third issue relative to the generation of handling actions at runtime, via a reasoning process. This last issue is introduced in order to complete the framework of an agent execution model robust to exceptional situations, but it essentially remains a track for future research.

The chapter is organized with a first overview section of the extended model, presented as an execution cycle. Section 5.2 elaborates on the acquisition of additional actions by the agent during the exception management process, and sec-
5.1 Extension of the execution cycle

The extended execution cycle of agents is presented in Fig. 5.1, with four levels. We describe them in turn in the following, focusing on the addition to the fundamental model.

**Figure 5.1: Execution cycle of an agent with incremental exception handling mechanisms**

**Fundamental model.** The upper part of Fig. 5.1 represents graphically the mechanisms of our agent execution model. The relevance and expectation filtering are formally represented in the previous model, as well as the state update and the interface to the decision process. The model also includes the handler selection and preparation, where handlers are actions that supersede other actions interrupted by exceptional conditions.

**Extensions to the cycle.** Table 3.1 introduced different classes of exceptions that agents can face in their executions (see page 46). As for agent exceptions, the fundamental execution model presented in the previous section deals with known
exceptions, with additional adaptive mechanisms for flexibility in the handling process. Unknown exceptions require additional mechanisms, primarily inspired by AI techniques.

Unknown exceptions are managed in the extended execution cycle by the additional handler search, evaluation, and generation functionalities. Whenever exceptional conditions are detected, the handler selection process is initiated with the following effects.

1. If a handler is found, the fundamental execution model is appropriate in this case.
2. If a handler cannot be found, the agent searches one, typically in an external repository \[61\].
   a) If such handler is found from external resources, the agent evaluates it and prepares it in case of positive evaluation.
   b) If no handler is available or the evaluation is negative, the agent attempts a generation, which always succeeds in returning a default handler (typically the failure of the activity, cleaning of the knowledge base, and information of the owner, as done in the execution model).

The elaboration of basic techniques to achieve these functionalities and their integrations in the cycle are the focus of the remainder of the chapter.

5.2 Acquisition of additional handling actions & discussion on generation mechanisms

If no handler is found in the selection stage, the agent encounters an unknown exception, i.e. the agent does not own any corresponding handling action. 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 [61, 100] 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. In general, the evaluation process is a complex issue and we elaborated a complete approach in some cases, detailed in the next section 5.3.

5.2.1 Acquisition of an additional handler

The search for a handler is an additional mechanism at the plan level (exception management) presented in Alg. 5.1. This search procedure applies in the environment a specialized operator called select, which concretely has a similar effect as a SQL select statement: It searches a target handler repository for handlers that satisfy some conditions. The condition is passed as the only argument of the procedure. The rational effect \( r \) is the target result that the searched handler should
produce. The condition could be extended with the activation context of a handler, but this element restricts the range of possible—and fortuitous—handlers that can be collected by the agent for the problem at hand. The algorithm assumes the definition of the evaluation procedure (see next section), and appropriate modification of the exception_plan procedure (see page 69) to lead the control flow to it whenever required. This modification is trivial.

```plaintext
Input: Agent state \( S \), Action \( action \), Time \( t \)
{ Select the handlers from the repository that satisfy \( action.r \) at least };
\( result_set \leftarrow \text{select}\{ h \mid action.r \vdash h.r \} \);
if \( result_set \neq \emptyset \) then
    { Appropriate handlers found, need for evaluation };
    \( validated_set \leftarrow \text{evaluate}(result_set) \);
    foreach \( h \in validated_set \) do
        if \( h \) is an operator then
            \( O \leftarrow O \cup \{h\} \);
        else if \( h \) is a protocol then
            \( Proto \leftarrow Proto \cup \{h\} \);
        end
    end
    { Retry to handle the exception };
    exception_plan(\( S, action, t \));
else
    { No handler found. Failure of the plan };
    if \( action \) is an operator then
        \( O^* \leftarrow O^* \cup \{action\} \);
    else
        { action is a protocol };
        \( Proto^* \leftarrow Proto^* \cup \{action\} \);
    end
    evaluate(\( S, plan, t_{now} \));
end
```

Algorithm 5.1: search(\( S, action, t \))

The collection of candidate handlers constitute the additional knowledge acquired by the agent, which should then evaluate each handler to decide which are appropriate and safe for its problem at hand. The algorithm loops the execution back into the fundamental model by exploiting the exception_plan and evaluate procedures adequately.

We note that Alg. 5.1 is presented differently from the procedures of the agent execution model. There should be two procedures (for applying the operator and receiving the result), but we merged them into one for presentation purpose. In other words, the above example assumes the operator is applied in a synchronous way.
5.2. ACQUISITION OF ADDITIONAL HANDLING ACTIONS & DISCUSSION ON GENERATION MECHANISMS

5.2.2 Discussion on 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 and attempts more than a direct failure report to the owner.

**Default handler.** The default handler of the execution model is to ‘ask for repeating’ a finite number of times until the agent can get an expected message, before considering the corresponding action has failed and reporting the case to the owner. This handling method can be interrupted by the *timeout* procedure if the execution lacks time for completion. The generated handler is parameterized with the expected message and put in the protocol enactment set *Proto* at the Handler preparation stage. For instance, the handler $h_{geo}$ generated to wait for the message transfer $transfer(arg[])$ within three execution cycles concerning the protocol $p$ is defined as follows. We shorten the syntax of a FIPA request message with the expression $request(transfer)$.

\[
\begin{align*}
h_{geo} := & \text{ request(transfer),} \\
& \text{ (transfer) } \\
& \text{ (request(transfer), (transfer) )} \\
& \text{ (request(transfer), transfer)) }
\end{align*}
\]

The agent then expects to receive the *transfer* message in one of the three steps of the handler, and then returns to the interrupter action. At each step, the agent ignores the message if it is not conforming with its expectations. Beyond the reception of three non-conforming messages, the handler is supposed to fail in turn, and the corresponding plan is reported as a failure, unless time allows re-planning.

**Discussion on complex handler runtime generation** 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 request ‘repeating’ a message seems appropriate as default. Ask for repeating a message causes no harm to the agent state and can maintain activities, provided time is available. The repetition-based handler is however insufficient whenever the exception has a real impact on the activity.
Two ways seem particularly relevant 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, such as the repository proposed by Klein and Dellarocas [61]. 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, and still satisfying activation context or rational effect conditions. 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. Our 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, which is compatible with the notion of expectation specified in our execution model. The hypothesis allows the agent to 'simulate' 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 shows the potential of abduction [107, 106, 58], but the technical issues of such approach are numerous and the present research remains in a premature stage.

5.3 Automated evaluation of handling protocols

Autonomous agents need the capability to evaluate which handlers best suit the activation context in a safe way. The evaluation stage is essential if the agent is to remain autonomous and to ensure that handlers obtained from external entities are safe and reliable. Handlers from untrusted, third-party knowledge bases may in fact lead agents to undesired behaviors or cause damages (e.g. inconsistent state). The aim of this section is to present an integrated approach to have agents reason about handlers and select the best ones with respect to the protocol they are executing. The approach focuses on a restricted case where the problems of untractability can be avoided. The illustrative application is therefore a sub-case of the case study.

We first describe the example and the assumptions underlying this extension of the model. Section 5.3.2 then presents a formal model of the handler evaluation problem that is exploited to detail our approach in section 5.3.3. Section 5.3.4 discusses the limitations of the proposed method, and relates it to existing work.

5.3.1 Running Example

The Store Order Protocol (SOP) represented in Fig. 5.2 describes a simplified version of the protocol from the case study, with the notable difference that SOP
is limited to one client and one store for presentation purpose.

In SOP, the client starts with ordering an item X to the store. The store may reply the item is not available (n-a) or propose a price Y. On reception of a price, the client can either cancel the order or pay Y. When the payment is completed, the store finally ships X. Note that the formal notation transfer(arg[]) is simplified to the name of the performative of the intended message for compactness of the formulas.

A number of ‘unexpected situations’ can occur in unrestricted environments, including timeout (not represented) infringement in replying to messages, malformed messages, improper sequences, and so forth. We will focus on a timeout infringement to present the evaluation approach. Typically, we describe the execution of the method when the store fails to ship the item on time: The client has to react appropriately.

We assume in the remainder of the section the following:

- Agents only transfer messages and do not exploit operators.
- The client executes only one SOP protocol (for presentation purpose).
- A timeout exception has been detected for receiving the ‘ship(X)’ message.
- The agent receives a non-empty set of handlers from a repository [61].

The approach is independent from these assumptions, which are introduced to focus on the proposed mechanisms.
5.3.2 Formalization of the problem

The evaluation of handlers relies on the knowledge of the agent with respect to known protocol only in this restricted case. The aim of this section is to introduce our model of the problem.

5.3.2.1 Knowledge Structure

The knowledge of interest for handler evaluation is a specific sub-set of the agent knowledge. We distinguish the problem-dependent handling knowledge ($D$) from the problem-independent one (PIK). First, $D$ is the operational knowledge of the agent for handling, represented as a $n$-tuple, where $n$ is the length of the corresponding knowledge (e.g. the number of beliefs, desires, and the current intention). In the running example, agents know the protocol, their current money, the state of their order, and some constraints, as shown respectively in the next formula.

$$D = \langle P, \text{Money}, \text{order}, (C, =, \prec) \rangle$$

$P$ is the aforementioned SOP, the ‘money’ is a numerical value, and ‘order’ is either $\text{nil}$ (order not shipped) or $X$ (with respect to SOP). Constraints in $C$ are predicate formulas on $D$, with a preference structure and the binary operators ‘=’ (no preference) and ‘$\prec$’ (total order). Any constraint must hold, unless a most preferred one already holds.

$$\forall c \in C, [c \text{ is true } \lor \exists d \in C, (c \prec d \land d \text{ is true})]$$

PIK represents what the agent knows about the current execution of the protocol. It is independent as it applies to any protocol executed by the agent.

$$\text{PIK} = \langle \Sigma, \text{next}(P) \rangle$$

$\Sigma ::= P.history$ is a notation for the representation of the protocol enactment introduced in the agent execution model. It is the actual course of execution, normally following the protocol, but expanding to unexpected situations whenever exceptions occur. $\text{next}(P)$ represents the next possible message transfers in the protocol from $\Sigma$, as defined on page 60 of the execution model.

5.3.2.2 Protocol graphical & formal representation

Protocol are tree structures that can be represented as graphs, as special case of Petri nets. The difference with Petri nets is that transitions have only one output arc (no need for concurrency model in the present work). The graphs are trees with a depth-first place and transition numbering. Thick-line places are terminal states in the protocol and form a set $E$. Fig. 5.3 depicts the representation of the SOP protocol.

During protocol enactment, places (circles) represent the state of the agent knowledge structure. The set of terminal states is $E = \{S_4, S_5, S_6\}$ in this example.
Transitions (squares) are message exchanges labeled with the type of message. The transitions allow to describe history of interactions Σ = (t_i)_{i≤n}, where n is the number of the last fired transition. An additional syntax is introduced, so that agents can reason explicitly on their roles in the protocol. This syntax precedes all messages in the protocol according to the stance of the agent (sender or receiver). T ::= !α | ?α. The exclamation mark declares the sending of message α, whereas the question mark is the reception of α. Fig. 5.3 shows how the client in SOP considers the protocol.

5.3.2.3 Semantics

The semantics of this model pertains to the evolution of the agent knowledge structure, when protocols execute. Only PIK evolves in this semantics, as D is merely initialization knowledge for the presented mechanisms. The next formula is the semantics over PIK.

\[
\text{exec}(P) : \text{Event} \times \text{PIK} \rightarrow \text{PIK}
\]

\[
(t_j, \langle \langle t_i \rangle_{i<j}, \{t_j\} \rangle) \mapsto \langle \langle t_i \rangle_{i<j}, \text{next}(P) \rangle
\]

\[
(e, \langle \langle t_i \rangle_{i<j}, \emptyset \rangle) \mapsto \langle \langle t_i \rangle_{i<j}, \emptyset \rangle
\]

The semantics states that Σ (first operand) stores the series of actually fired transitions during the execution. The successor transition (second operand) points to the next transition to fire or none if a terminal state is reached. This semantics covers only the case where there is only one successor to save space and still show the essential mechanism. Generalization to any set of successors is straightforward.

5.3.2.4 Exception at run-time

Fig. 5.4 shows a graph that represents the enactment of P after the client has sent the message ‘pay(Y)’ for the transition t_3, and a timeout event t_{ex} has occurred. In this situation, the agent has detected an exception, as it was expecting a ‘ship(X)’ message from the store, according to the definition of P in Fig. 5.3.

The dashed arrow and circle represent the expected state in accordance to the protocol, where S_4 is the state of the agent knowledge when the ‘ship(X)’ message is delivered. Handling the timeout infringement exception is therefore finding a handler that ‘bridges’ S_{ex} to S_4, i.e. a handler that meets the requirements in
terms of rational effects. Among the different handlers available for this bridge and satisfying the target rational effect, the next section presents how the agent evaluates and chooses among them.

5.3.3 Handler evaluation

Given \( n \) handlers that bridge \( S_{\text{ex}} \) to \( S_4 \) (previously awaited rational effect), the agent can choose only one to enact and must therefore evaluate the best choice in its execution context. Fig. 5.5 shows the case where \( n = 3 \). The following method allows the agent selecting the best candidate paths, i.e. handlers with the best evaluation.

The rationale of the handler \((H_{1i})_{i \leq 1}\) is for the client to remind the ship commitment to the store. The handler leads the client to poll the store with a new timeout, and the store should reply with the actual shipment. Handler \((H_{2i})_{i \leq 2}\) reports a delay, expects a delay message, and finally the awaited shipment. Handler \((H_{3i})_{i \leq 2}\) ask for compensation, expects it and the shipment.

The handlers are all appropriate as they lead the agent back to \( S_4 \), originally expected. They however have different characteristics that lead to prefer some above the others.
5.3. AUTOMATED EVALUATION OF HANDLING PROTOCOLS

5.3.3.1 Evaluation criteria

The evaluation pertains to value handlers with two criteria, namely their structures and the verification of constraints. The structure of a handler matters as complex networks can be more flexible but introduce more uncertainty on the outcomes. Verification of constraints is essential for the agent to guarantee that any stage of the handler does not cause any undesired changes. The value of a handler is a single number, and the best handlers are the ones with the minimal value.

The structure of handlers is evaluated against the following table.

<table>
<thead>
<tr>
<th></th>
<th>Uncertainty rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending (!)</td>
<td>(0 &lt; a &lt; 1)</td>
</tr>
<tr>
<td>Receiving (?)</td>
<td>(0 &lt; \beta &lt; 1)</td>
</tr>
<tr>
<td>Branching</td>
<td>Maximal uncertainty of the branches</td>
</tr>
</tbody>
</table>

Table 5.1: Structure evaluation

The costs of sending and receiving are abstract values \(\alpha\) and \(\beta\), as the purpose is to have a relative ranking of the handlers only. When a handler contains different branches, the meaningful cost is necessarily the maximal one, i.e. worst-case scenario.

The verification of constraints relies on \((C, =, \prec)\), the preference structure over constraints. The preference allows to represent constraints as a tree, where roots are the preferred constraints and leaves the less preferred ones. Typically, \(C = \{C_1, C_2, C_3, C_4\}\) and \(C_1 = C_2, C_2 \prec C_3, \text{ and } C_3 \prec C_4\) can be represented as shown on Fig. 5.6.

The tree shows that \(C_1\) and \(C_2\) must be verified, while others are preferable but not compulsory. The evaluation values each constraint according to its level in the tree. The preferred constraints have lower scores (bonus), since the best handlers have the lowest values. We note \(\max(C)\) the set of constraints with the maximal score (0), and we assume the availability of the unary function 'score(c)' that outputs the score of any constraint \(c\).
5.3.3.2 Evaluation algorithm

The evaluation is conducted in three stages as presented in Alg. 5.2.

```
Input: Candidates, C
Output: selection
selection ← ∅;
best ← 0;
foreach handler in Candidates do
  scoreS ← 0, scoreC ← 0;
  if ∀ c ∈ max(C), c ⊢ handler then
    { The handler verifies all root constraints }
    scoreS ← structure(handler);
  foreach c ∈ C \ max(C) do
    if verify(handler, c) and score(c) is minimal on the constraint tree branch then
      scoreC ← scoreC + score(c);
    end
  end
  if (scoreS + scoreC < best) or (best = 0) then
    best ← scoreS + scoreC;
    selection ← ∅;
    selection ← selection ∪ {handler};
  else if scoreS + scoreC = best then
    selection ← selection ∪ {handler};
  end
end
```

Algorithm 5.2: Evaluation algorithm

For each candidate handler, the algorithm first verifies that all root constraints hold (line 6). If a handler violates any constraint, it is discarded as it would be hazardous for the agent to attempt executing it. Handlers that verify root constraints are further evaluated by valuing 'scoreS', the score over the structure, and 'scoreC', the score for the preferred constraints. The case of the score structure is illustrated in the example of the next section. The score of the constraints aims at finding if the handler verifies preferred constraints. If such a verification is found, the algorithm assigns the minimal cost along the constraint tree branch (lines 10-12). For example, the verification of $C_3$, but not $C_4$, yields 'scoreC=-1'. The algorithm does not need to add the score of the root constraints, as all handlers must verify the root constraints at this point and the aim is a relative comparison of the handlers. If the total score of the handler is lower than the best one or if the best score is 0 (initialization, scores are strictly positive), then it becomes the best score and the output list of handler is replaced by the current one. If the score is higher,
the handler is discarded, and if the score is the same, the handler is added to the output list.

On termination of the algorithm, the agent is handed in with a list of handlers that have the same evaluation. It is free to choose among them, with the same degree of success relatively to the evaluation process.

5.3.3.3 Example

Constraints in the running examples can be the following $C_1$ and $C_2$:

$C_1 = \forall i < n, \text{Money}_i = \text{Money}_{i+1}$
$C_2 = \exists i < n, \text{Money}_i < \text{Money}_{i+1}$
$C_1 \prec C_2$, so $\text{score}(C_1)=0$ and $\text{score}(C_2)=-1$

According to the algorithm and the structure evaluation table (Tab. 5.1), the following results are obtained for the three handlers of the example.

<table>
<thead>
<tr>
<th>Handler</th>
<th>score$S$ + score$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(H$_1$)</td>
<td>$\alpha + \beta + 0$</td>
</tr>
<tr>
<td>(H$_2$)</td>
<td>$\alpha + 2\beta + 0$</td>
</tr>
<tr>
<td>(H$_3$)</td>
<td>$\alpha + 2\beta + (-1)$</td>
</tr>
</tbody>
</table>

Table 5.2: Evaluation of the handlers in the example

In consequence, (H$_1$) is better than (H$_2$) due to a simpler structure, and (H$_3$) is preferred over (H$_1$), since it allows to meet a preferred constraint. The algorithm outputs (H$_3$) only in this example, and the agent will select it for the subsequent exception handling stage. If the output was a set, the agent is free to choose any handler from the output with the same ‘guarantees’, but different strategies. The rationale of choosing (H$_3$) at the end of this algorithm is that this handler is not too complicated structurally, and it allows the agent to earn money in the handling (compensation), which is relevant for this particular problem.

5.3.4 Discussion

Presented approach. The evaluation approach allows agents to select the best handlers in their context, relative to individual and problem-dependent criteria (i.e. the constraints). As for the complexity of this method, we can show that it is linear in both $n$, the number of handlers to evaluate, and $h$, the length of the longest handler. In fact, each stage of the algorithm is bounded by the product of these values, so that the complexity is $O(nh)$.

The evaluation of handlers relies on valuing the structure and the verification of constraints. The presented approach does not refer to cases when handlers contain loops. Relying on the unfolding property of Petri nets, we consider that loops can be deployed along branches of the protocol or handler tree. Loops necessarily
increase the length of a handler, and their scores are therefore usually higher than handlers without loop. This result is sound with the fact that long handlers have a higher uncertainty of success and that shorter ones should be preferred. As for the verification, more complex and accurate scoring methods can be designed, but the current approach has the advantage to be simple and computationally economical.

**Existing work and possible alternatives.** The evaluation of exception handlers is akin to the work on planning, workflows and related applications to web services. Planning requires techniques for plan repair or plan extension, without necessarily re-planning from the beginning [106, 125]. The difference with interaction protocols is that protocols are reified shared artifacts that serve coordinating agents, as modeled in our agent architecture. A plan can be a shared knowledge, but the structure is flexible and more volatile as it can be re-planned dynamically. A protocol enactment can be thought of as the execution of a plan however, and that is why the two notions are close and their syntax have similarities in our model.

Research on workflows has seen a number of endeavors to deal with exceptions, especially in Software engineering [9, 62, 69]. The approach with protocols is very close, as can be observed with the common formal representations that are exploited. The particularity of interaction protocols is that agents reason about them, so that the process enactment is at a meta-level and agents tolerate deviations from the protocol. Workflow systems tend to introduce such meta-level reasoning on a workflow enactment to deal with exceptions, e.g. [9], and the approach presented in this section can be seen as an AI solution to the problem.

### 5.4 Complexity analysis

Given the algorithms presented in this chapter, the purpose of the complexity evaluation is to measure the overhead computational cost of the exception management model and orders of magnitude for the extensions. Practical evaluation based on this analysis are conducted in chapter 7 on the validation of the model.

#### 5.4.1 Notations

The cost of the Decision process stage, as shown in the execution cycle of Fig. 5.1, page 76, 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 $N_{Gen}$. In the case of the approach presented in the previous section, $N_{Eval}$ is valued $O(nh)$ for example, where $n$ is the number of handlers to evaluate, and $h$ is the length of the longest handler. Let us also note $n_{act}$ the size of the operator and protocol enactment sets $\text{Proto}^* \cup O^*$, representing the number of protocols and operators exploited by the agent, and $n_k$ the number of protocols and operators known to the agent (as potential handlers during execution).
5.4. COMPLEXITY ANALYSIS

5.4.2 Complexities

In the overall agent execution model, the complexity of each box depends on the number of expectations generated by the agent. A typical execution cycle generates a number of expectations (cardinal of the set returned by the \textit{next} operation) that is proportional to the number of actions in execution, which entails a complexity of $O(n_{act})$. This complexity assumes that protocol definitions are ‘reasonable’ however, i.e. the complexity of the protocol tree is linear. This assumption is verified in practice, since the alternative message choices are limited. We have the following costs for each algorithm.

- $O(n_{act})$ for generation of expectations
- $O(n_{act})$ for Relevance Filter and Expectation matching
- $\max(N_{DP}, O(n_{act}))$ for Decision Process
- $O(n_{act})$ for State Update
- $O(n_{k})$ for Handler Selection
- $O(n_{act})$ 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 repositories. The communication cost is however increased, but it does not participate in the computational one we are evaluating here. The following table 5.3 compiles the different overhead costs depending on the execution type.

<table>
<thead>
<tr>
<th>Execution type</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>No handling</td>
<td>$N_{DP}$</td>
</tr>
<tr>
<td>With handling</td>
<td>$N_{base} = \max(N_{DP}, O(n_{act}))$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overhead cost over $N_{base}$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Known exception</td>
<td>$\max(O(n_{k}), O(n_{act}))$</td>
</tr>
<tr>
<td>Unknown exception, Search &amp; Evaluation</td>
<td>$\max(N_{Eval}, O(n_{k}), O(n_{act}))$</td>
</tr>
<tr>
<td>Unknown exception, Search, Generation</td>
<td>$\max(N_{Eval}, N_{Gen}, O(n_{k}), O(n_{act}))$</td>
</tr>
</tbody>
</table>

Table 5.3: 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. 5.1). If we assume that in practice agents are expected to run a low number of protocols simultaneously, a cost of $O(n_{act}^2)$ (assuming $O(n_{act}) \sim N_{DP}$) 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 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 handling actions that the agent knows is a costly factor as soon as appropriate ones 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. $N_{\text{Eval}} < O(n_k)$. However, we think that a robust evaluation is costly, especially to guarantee the handlers are acceptable.

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

In the era of object-oriented software engineering, agents and multi-agent systems gain focus as promising abstractions to structure and (purposefully) design complex software systems. An engineering framework is consequently useful to guide the design of agents, when they are deemed as appropriate for the target system. The purpose of this chapter is to elaborate on the formal agent execution model to derive a practical engineering framework. The framework provides software designers with guidance and focus on (i) the services included in the execution model and (ii) the tasks to achieve in order to design a ‘robust agent’. The engineering framework leads to a practical software architecture of an agent that can serve as a basis for system development.

Section 6.1 presents the engineering framework with focus on the tasks of the designer in building agents and the automation provided by the model. Section 6.2 then presents the software architecture for agents and its relation to the model.

6.1 Framework

Our engineering framework aims at defining services provided by our model to the agent designer, and to identify precisely the set of required inputs from the designer to build an agent integrating exception management capabilities. This section first reviews the services provided by the model, and then compiles the necessary tasks to design a working agent model appropriate for implementation.

6.1.1 Services of the framework

The framework provides the following design interface to exploit the services of the model and create agents:

- Structured knowledge base that allows the designer (and owner at runtime) to assign goals, protocols, and operators to the agent.
• 'Connectors' to integrate appropriate decision, planning, and reporting modules into the framework.

• Action selection scheme based on the definition of activation context and rational effect for the actions of the agent.

The main activity of the designer in the framework is to determine the appropriate contents of the agent knowledge, in addition to the representation scheme. Approaches in Computational logics such as KGP require the contents being in predicate logic [58]. Approaches for reactive agents such as the Subsumption architecture require an implicit representation 'hard-coded' in the agent execution mechanisms [10]. Other approaches often require knowledge as variables in imperative languages [54, 68, 132]. Our model is compatible with all these approaches, provided appropriate implementations and despite the formal model in Computational logic, so that the 'connectors' of the framework can accept any knowledge representation in theory, and plug to appropriate modules according with system requirement.

Plans are not part of the engineering framework, since their management is transparent to the designer. The choice of a planning module is the actual design choice in the framework that allows dealing with the planning capabilities of the agent. The model then manages plans appropriately to achieve the goals of the agent and handle exceptional situations.

The framework also sets forth the action selection scheme of the agent. This scheme is provided by the framework, and it overrides the one usually integrated in the decision modules, such as the mechanisms in KGP [58] or ANA [71]. The model selection scheme provides similar functionalities than the ones in decision modules, but it focuses on the exception management approach based on expectations and rational effects. We think that dedicated implementations of the model with a given decision module would allow exploiting the module action selection mechanism in place of the model one.

6.1.2 Design tasks

In the light of the services provided by the framework, the designer should focus on the tasks compiled in the following table for creating an agent with exception management capabilities.

Existing engineering framework such as KGP, Jade or Jason provide similar amounts of tasks to be accomplished by the designer. Our framework integrates however the management of exceptions on behalf of the designer. Experimental analysis in chapter 7 will detail the contribution of our framework by comparing it to alternative approaches in engineering agents with exception management capabilities.
Table 6.1: Tasks for designing agents with exception management capabilities

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Establish the agent knowledge</td>
<td>Choose the protocols, operators, and domain application knowledge necessary to fulfill the requirements</td>
</tr>
<tr>
<td>Connect decision, planning, &amp; reporting modules</td>
<td>Provide necessary connection interfaces &amp; modules</td>
</tr>
<tr>
<td>Constrain protocol and operator execution</td>
<td>Decide the rational behavior of the agent by the selection of activation contexts and rational effects</td>
</tr>
<tr>
<td>Integrate additional mechanisms</td>
<td>Additional design tasks related to handler selection, evaluation, and generation</td>
</tr>
</tbody>
</table>

6.2 Software Architecture

The architecture of the agent aims at setting forth the architectural elements that correspond to the execution model, encompassing the extensions of section 5.1 as ‘placeholders’ for the late design and analysis phases of the agent development process. 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' [7]. 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 extended execution cycle.

6.2.1 Abstract architecture

Figure 6.1 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.

6.2.2 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 [133]). 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 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 [98, 58], 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 provides the decision and planning modules or possibly others, including the preceding examples or PRS, MANTA, etc. [37, 21], 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. In compliance with the execution model, 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.

Actuation. The action command finally entails the generation of expectations 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 expectations in the internal representation. In the end, the command is applied in the environment by the actuator function.

6.2.3 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 and the second column compiles the elements of the architecture. 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.

The table shows 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 [105]. 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 ($S$). One element per compound.

**Exception logic.** The remaining elements, with at least one per compound (the internal representation is also part of it in $S$).

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.
Seven

Experiments and 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 onward. 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.

7.1 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 environment, 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 [49]. 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 fundamental agent execution model 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.

### 7.1.1 Scope of the EMS implementation

The EMS implementation covers the extent of the fundamental execution model, without the extensions. Fig. 7.1 shows the coverage of the implementation over the model. The extended 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 7.1: Coverage of the implementation (plain lines) over the execution model (plain and dashed lines)](image)

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 handling action capabilities 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.

\[
\text{RefuseDelay} = (\text{inform}(\text{RefuseDelay}))
\]

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 handler allows then to return immediately to the interrupted protocol by terminating the handler. Its activation context is not set, so that the agent can invoke this protocol in any context. The rational effect is the same as the reject performative defined in the example. In other words, the result of the handler is to consider that the requesting agent is rejecting its participation in the negotiation process. This rational effect allows the agent to continue the protocol, unless there is no more participant, in which case the protocol fails and entails the failure of the corresponding plan and goal.

The pattern to select the handler is presented in the following formula, based on the message pattern (c.f. formula 4.2.1.1, page 57).

\[
(\ldots) \text{inform, } \text{delay} = (\backslash d +)
\]

Messages that trigger the RefuseDelay handler are therefore messages with an inform performative and a content that matches the regular expression \(\text{delay} = (\backslash d +)\). For instance, the expressions \(\text{delay} = 500\) and \(\text{delay} = 12000\) are recognized by this expression, whereas \(\text{delay} = 5.00\) and \(\text{delay} = \text{Tonight}\) 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. 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.
7.1.2 Experimental protocol

The series of experiments conducted in this research so far followed the process depicted in Fig. 7.2.

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 \texttt{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. 7.2. The log files are processed to extract statistical information and compared to evaluate the overhead in practice.

7.1.3 Technical details

The code of the experiments was written in Java 5.0 for portability and efficiency reasons (due to the increase in native code in the version 5 of Java). 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 debugging, 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. 7.3 shows how part of the agent architecture has been implemented. The figure shows the complete code of the \texttt{handlerSelection} method, which aims at implementing the functionality of the same name in the execution model and software architecture.

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 \texttt{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 7.3: Sample code from the experiments: Agent method handlerSelection

7.2 Qualitative Analysis and Comparison

7.2.1 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.

### 7.2.2 Comparison

Table 7.1 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.

<table>
<thead>
<tr>
<th></th>
<th>Plain</th>
<th>Sentinels</th>
<th>EMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation of concerns</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Maintainability of exception handling code</td>
<td>Low</td>
<td>Med</td>
<td>High</td>
</tr>
<tr>
<td>Robustness to autonomous behaviors</td>
<td>Low</td>
<td>Low</td>
<td>Med/ High</td>
</tr>
<tr>
<td>Tasks for engineering handling code</td>
<td>Ad hoc</td>
<td>Sentinels and Protocols</td>
<td>Handlers</td>
</tr>
</tbody>
</table>

Table 7.1: 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. 5.1, page 76) and a part for the handling (the bottom part of Fig. 5.1).
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 maintainability 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 handling actions at runtime automatically. Maintenance can thus target the mechanism or—individually—actions (and their activation contexts) 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 the tasks to the creation of handlers only. Handlers design corresponds to protocol design with decisions about appropriate activation contexts, so that we can expect the task to be easier than for the sentinel approach.

### 7.3 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.

#### 7.3.1 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. 7.4. The figure shows a typical time-series of the performance of an agent in the market. The
7.3. EXPERIMENTAL RESULTS

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

![Figure 7.4: Number of execution cycles completed by agent 'Machine Assembler 1' - No EMS](image)

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. 7.5 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.
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. 7.6 and Fig. 7.7 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 7.2 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
7.3. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>Type</th>
<th>Max Average</th>
<th>Min Average</th>
<th>In Stationary Interval Max delta</th>
<th>Max</th>
<th>Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception-free</td>
<td>3.57</td>
<td>1.0</td>
<td>0.09</td>
<td>1.09</td>
<td>1.0</td>
</tr>
<tr>
<td>EMS</td>
<td>2.13</td>
<td>1.0</td>
<td>0.04</td>
<td>1.04</td>
<td>1.0</td>
</tr>
<tr>
<td>Ratio (±10−2)</td>
<td>0.568</td>
<td>1.0</td>
<td>2.25</td>
<td>1.05</td>
<td>1.0</td>
</tr>
<tr>
<td>Inverse (±10−2)</td>
<td>1.761</td>
<td>1.0</td>
<td>0.45</td>
<td>0.95</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 7.2: Comparison of the performance characteristics

![Figure 7.6: Average number of execution cycles completed by agents over 100ms periods - Exception-free](image)

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
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 a 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 5.3 (page 89) leads to the following estimation, with $N_{DP}$ the complexity of the reference system, $O(n_k)$ the complexity for Handler Selection, and $O(n_{act})$ for Handler Preparation. The complexities are related to the execution time per cycle, so that the estimation is
7.3. EXPERIMENTAL RESULTS

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

<table>
<thead>
<tr>
<th>Execution type</th>
<th>Theoretical Cost</th>
<th>Order (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception-free</td>
<td>(N_{DP})</td>
<td>(10^3 \ (2345))</td>
</tr>
<tr>
<td>EMS</td>
<td>(N_{base} = \max(N_{DP}, O(n_{act})))</td>
<td>(\max(10^3, O(1)))</td>
</tr>
<tr>
<td>Known exception</td>
<td>(N = \max(O(n_k), O(n_{act})))</td>
<td>(\max(O(1), O(1)))</td>
</tr>
<tr>
<td>Total estimation</td>
<td>(N_{base} + N)</td>
<td>(10^3)</td>
</tr>
<tr>
<td>Measured Value</td>
<td></td>
<td>(10^3 \ (5081))</td>
</tr>
</tbody>
</table>

Table 7.3: 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.

7.3.1.1 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. 7.8 and Fig. 7.9 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.
The curves of the agent in the two situations are similar on these runs. In the context of the whole system, Fig. 7.10 and Fig. 7.11 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
7.3. EXPERIMENTAL RESULTS

Figure 7.10: Average of the capitals of agents over time (red), and Bezier approximation for the average and each agent (other colors) - No EMS

Figure 7.11: 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. 7.10 shows that the monotonicity changed at most twice, and for only one agent. In the EMS version, Fig. 7.11 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. 7.12 and Fig. 7.13 show the occurrences of exceptions over time and the superimposition with the agent capitals respectively.

![Graph showing the occurrences of exceptions over time](image)

Figure 7.12: 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: Start handler...
```
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.

7.3.1.2 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. 7.14 and Fig. 7.15 show the graphs of the two phases over time.

![Figure 7.14: Average number of relevance rules generated by agents over time (red), and Bezier approximation for the average and each agent (other colors) - With EMS](image)

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 in the implementation. 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. 7.7), 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 and expectation elements that serve in the EMS mechanism demand a significant computation. As a result, the consequent cost can be controlled by optimizing the management of
7.3. EXPERIMENTAL RESULTS

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

7.3.2 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 7.4 and 7.5 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.

Figure 7.15: Average number of expectations rules generated by agents over time (red), and Bezier approximation for the average and each agent (other colors) - With EMS
### Table 7.4: Comparison of the performance characteristics

<table>
<thead>
<tr>
<th>Approach</th>
<th>Max</th>
<th>Min</th>
<th>Max in Stationary Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception-free</td>
<td>3.57</td>
<td>1.0</td>
<td>1.09</td>
</tr>
<tr>
<td>Plain</td>
<td>3.37</td>
<td>1.0</td>
<td>1.08</td>
</tr>
<tr>
<td>EMS</td>
<td>2.13</td>
<td>1.0</td>
<td>1.04</td>
</tr>
</tbody>
</table>

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 elements, 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.

### Table 7.5: Average computational cost of an agent cycle in terms of execution time

<table>
<thead>
<tr>
<th>Approach</th>
<th>Time (±10−2s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception-free</td>
<td>2.35</td>
</tr>
<tr>
<td>Plain</td>
<td>2.35</td>
</tr>
<tr>
<td>EMS</td>
<td>5.08</td>
</tr>
</tbody>
</table>

### 7.4 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 elements can allow to reduce this cost, such that specialized versions of the EMS might be
adapted to resource-limited platforms.
Part III

Uniφ: Building Agent Systems
Robust to Exceptions
Exception management in agent systems intervenes at two complementary levels, namely the agent- and system-levels. We have explained in the previous parts that our work aims at the agent level, so that agents can cope with exceptional situations by themselves, without relying on external support in the first place. One of the motivations for this work was that existing work focused primarily on the system level, introducing specific services to support agents in case of exceptions. The issue of these approaches is that agents can lose any support whenever these services are not available. In other words, the system robustness to exceptional situation depends on a sub-set of agents too tightly, thus limiting the potential effect of decentralized control. In addition, system-level approaches tend to override agent autonomy by taking over control of agents in case of exceptions.

The agent-level approach compensates the system-level issues by endowing each agent with exception management capabilities. Agents can rely autonomously on their sole capabilities to react to exceptions. The agent-level approach is however known less efficient than a system-level one. We can show analytically and by analogy with distributed system research that a system-level management of exceptional situations is more efficient than at the agent level (see the Guardian approach [82]). For example, agents may face circular wait deadlock situations during their interactions. An agent-level management of this exceptional situation yields two issues: (i) How does an agent realize the deadlock situation?, (ii) How to release the situation efficiently? Both issues can be managed at the agent level with appropriate mechanisms based on our execution model. It is however clear that a system-level mechanism would be efficient and straightforward to design for detecting and resolving the case.

In short, the complementarity of the agent- and system-levels is essential to balance advantages and limitations of each approach. Fig 8.1 shows two competing dimensions in exception management approaches: Efficiency on one axis, and the
autonomy of the agent paradigm on the other. Leveraging the complementarity leads to the Uniφ framework presented in this chapter.

![Diagram](image)

**Figure 8.1**: Uniφ approach as a complementarity of the agent- and system-level exception management systems

Fig. 8.1 shows the relative position of the different approaches on the efficiency-autonomy domain. The purpose of the present part is to combine the agent- and system-level main solutions into a new approach that is both efficient and preserving autonomy. Uniφ aims at addressing the performance issues of our agent execution model in some exception management cases by relating it practically with the sentinel approach [49]. In addition, Uniφ aims at adapting the sentinel approach so that it eventually becomes compatible with the preservation of agent autonomy, thus gaining to be applied to open heterogeneous systems where the designer does not necessarily know in detail the execution environment of the agents.

The chapter structure relies on our proposal to make sentinels ‘autonomy-ready’, i.e. the ability to provide their services while still preserving the autonomy of application agents. The idea relies on an agent architecture and interaction mean, so that agent decide *themselves* what information to disclose to sentinels. Section 8.1 presents tag interactions and the ‘softbody’-based architecture to have agents choose the information they agree to expose in the system. Section 8.2 then details a MAS model that includes agents based on our execution model and the softbody architecture, along with sentinel agents and appropriate interaction protocols. This approach on MAS combines the different approaches into the Uniφ framework. Section 8.3 finally discusses the framework advantages and limitations.
8.1 Tag interactions and softbody

8.1.1 Rationale of the interaction model

The main issue with sentinels is the possibility to control application agents and modify their internal states. This remote control is an inappropriate interaction pattern between sentinel and agent, where communication should be modeled as message-passing without tight control. Agents cannot preserve their autonomy with such external control, notably in open and heterogeneous systems. The sentinel approach can however be utilized if another interaction mean with application agents can apply without infringing autonomy. Tag interactions addresses this issue, inspiring from existing interaction schemes, such as indirect (e.g. overhearing and stigmergy) [59, 89], implicit [123], or opportunistic interactions [5].

Tag interactions rely on agents exposing information about them, so that other agents can freely read them, without infringing autonomy. Doing so, sentinel agents can read appropriate information about agents, and then send advices as usual messages to inform application agents about exceptional situations and potential solutions. The information is published on a distinctive publicly accessible software component of an agent architecture that we named the ‘softbody’. Agent can then autonomously decide their reactions using the messages from sentinels.

Tag interactions entail two distinctive issues. First, the public information exposed on the softbody is readable by sentinels that observe actively agents in the system, in the same way as execution monitoring [59]. The work of Gutnik on the monitoring selection problem shows that the choice of agents to monitor is not trivial [48]. Passive observation can complete active observation by notifying sentinels with changes in the system, which can lead to detect exceptional conditions. Passive observation relies on an event notification sub-system that needs to be integrated in MAS (with examples in related research [76, 138]. The second issue is that agents may publish incorrect information on their softbody, due to errors in their code or malicious intentions. A mechanism should therefore ensure that agents publish ‘acceptable’ information (i.e. proper type and range), without infringing autonomy, thus ruling the execution of tag interactions.

The two issues raised by tag interactions are addressed by the introduction of a ‘software environment’ [24, 92] responsible for enacting passive observation and ruling tag interactions.

8.1.1.1 Agent and softbody

Agents are now endowed with an explicit boundary named softbody that exposes:

- **Sensors** to receive information from the environment
- **Actuators** to send information to the environment
- A **public state** of the agent, observable in the environment
Exception Management with Autonomy-Ready Sentinels

Agent internals are explicitly hidden from other agents, 'encapsulated' in the soft-body boundary. This computational body features sensors and actuators for interactions with other agents and entities in the MAS [105]. The softbody also exposes a public state of the agent, which contains information about the agent that can be sensed by others in the environment. Concretely, the public state is a list of variables named tags whose values reflect agent internals. The contents and types of information in the public state can be configured by the designer or dynamically by the system to define what can be observed about each agent. For example, we can infer in a discussion that someone may be lying when we observe her to blush, i.e. her public state exposes a change to a 'red skin color'. On the other hand, system designers might choose to prevent such an observation in a software auction system to avoid collusion means by body signals [102]. Public state content information can indeed influence agent reasoning processes and consequently their interactions, notably in the case of the sentinels of the system.

The separation between the internals and the softbody is an architectural mean to preserve the autonomy of agents. Interactions are performed through the softbodies as interfaces, so that sending and reception of information are decoupled from internal process. Agents can decide autonomously in the execution model how to process information.

8.1.1.2 Tag interaction, agent, and environment

Definition: We define tag interaction as follows;

**Definition: Tag interactions**

Tag interaction is a set of mechanisms that models interactions based on the public state: (a) the expression of public state, (b) the sensing of public state, and (c) the monitoring and fortuitous propagation mechanisms.

Tag interaction mechanisms expose and sense public states with (a) and (b). We distinguish two types of propagation mechanisms to describe different situations in (c). **Tag monitoring interaction** is related to active observation, when sentinels collect information about others. This active inquiry mode is initiated by an agent and contrasts with the passive **tag fortuitous interaction**. This second type functions as a call-back mechanism: Sentinels receive information about the public states of application agents without explicitly requesting for it. Fig. 8.2 depicts the two types of tag interaction.

The left part of Fig. 8.2 shows the observing agent acting so as to sense the public state of the observed agent. The right part shows that the public state of an observed agent is spread out to the sensor of any other agent such as sentinels.

**Environmental Requirements.** Observing an agent means the public state is 'readable', such as on the left of Fig. 8.2. But in usual agent approaches, reading
actions are realized by contacting the observed agent that consequently becomes aware of this observation. The environment is a third party that can provide mechanisms to perform this function independently from agents. Tag fortuitous interaction emphasizes the previous argument, since the agent does not trigger an information transfer, as on the right of Fig. 8.2 where the environment delivers public state information. Finally, tag interaction requires an environment as a regulating entity [129]. The public state is a feature of the softbody that lets malicious agents fake attitudes or change the state of other arbitrarily if no regulation is enforced. The environment can enforce correctness of public states against system rules. Tag interaction requires from the environment the following support, and our model aims at providing them.

- Enact monitoring interactions
- Enact fortuitous interactions
- Regulate interactions

This support needs to be provided transparently to avoid introducing into the agent any complexity related to the environment responsibilities. The model abstracts this issue by only specifying what are the environment responsibilities. Implementation issues lie at another level of analysis. The environment might be centralized or distributed depending on application requirements.

8.1.1.3 Environment Model

Our definition of environment specializes a generic version [129], notably based on the work of Russell and Norvig [105], and Ferber [24]. It focuses on the salient characteristics of the environment for tag interaction.
CHAPTER 8. EXCEPTION MANAGEMENT WITH AUTONOMY-READY SENTINELS

The environment of a MAS is the entity where agents exist and that:

- Maintains the system topology
- Maintains mapping information of agents on the topology
- Performs tag interaction mechanisms
- Defines and enforces tag interaction rules

The environment is a stateful entity that defines and maintains a topology of the system, which can be spatial or abstract as in many simulations [65], network domains, file systems, or web-sites [6]. The environment maintains information about agent situations in the system topology to manage information delivery and regulate their interactions. This information is only related to softbodies. That is, the environment does not need to deal with agent internals, since they are encapsulated into agents and an access by the environment to internals would violate the autonomy assumption. The environment also mediates tag interactions, which refers to public state evolution and change notification. Furthermore, the environment applies rules in the MAS to enforce certain public state values. Rules define ranges of possible values, so that system states remain consistent for the application. Rules also define how public state information is spread out in the system, for instance by defining a range of interaction [74, 96]. These rules allow specifying change notification strategies to control the amount of tag information that is exchanged in the system (which can cause a significant cost variation as illustrated with the example application in the next chapter).

The definition meets the requirements we defined for tag interaction.

- Interaction mediation enacts an observation framework whereby the environment delivers observable events.
- Fortuitous events delivery is configurable by specific rules.
- Regulation is enforced by the rules while mediating interactions.

We notice that the environment has no deliberative capability and no decision power. The environment merely accomplishes its responsibilities in strict compliance with rules defined by design.
8.1. TAG INTERACTIONS AND SOFTBODY

8.1.2 Formalization

8.1.2.1 Agent

Our formalization of an agent follows from the execution model detailed in chapter 4, with appropriate extension for supporting tag interactions.

\[ Agent = (\psi, \phi, INF) \], where \( \phi = (S, A, P_s) \) \hspace{1em} (8.1) \]

First in this formula, \( \psi \) is a variable that encompasses the complete agent execution model proposed in this document. In addition, \( \phi \) represents the state of the softbody. A pair \((\psi, \phi)\) then represents a complete state of the agent. \( \psi \) ranges over the state space \( S_{\psi} \), and \( \phi \) ranges over \( S_{\phi} \). The softbody \( \phi \) is further developed into a 3-tuple where \( S \) is the set of sensors of the agent, \( A \) the set of actuators, and \( P_s \) the public state, which can be a set of variables in predicate logic.

The last element is \( INF \), a set of two reaction rules \( INF_{\psi} \) and \( INF_{\phi} \) to determine the evolution of agent states \((\psi, \phi)\) on change of internals or softbody respectively. For any \( \psi \) and \( \phi \):

\[
\begin{align*}
& (\psi, \phi) \text{ and } \psi \rightarrow \psi' \rightarrow \phi \rightarrow \phi' \rightarrow INF_{\psi} \\
& (\psi, \phi) \text{ and } \psi \rightarrow \phi \rightarrow \phi' \rightarrow INF_{\phi}
\end{align*}
\]

The operational semantics of \( INF_{\psi} \) expresses the evolution of the state \((\psi, \phi)\) after the evolution from \( \psi \) to \( \psi' \). The result of \( INF_{\psi} \) is the evolution of the softbody to reach \( \phi' \). The agent final state then becomes the pair \((\psi', \phi')\). For instance, when an agent wants to open a door, it first intends \((\psi, \phi)\) and then acts \((\phi, \phi')\) to complete its intention as a coupling with the environment. \( INF_{\phi} \) expresses similarly the evolution of internals due to the evolution of the softbody, e.g. input on sensors. \( INF \) operators explicate how internals and softbody are linked by a cause to consequence relation. What is modified along the evolution is application-dependent and relies on instances of the model.

8.1.2.2 Environment

Our formalization of the environment follows from the definition of section 8.1.1.3:

\[ Environment = (\Omega, \Phi, TRANS) \] \hspace{1em} (8.4) \]

A complete state of the environment is a pair \((\Omega, \Phi)\). \( \Omega \) is a 2-tuple, representing the environment internals:

\[ \Omega = (Topology, Rules) \] \hspace{1em} (8.5) \]

where \( Topology \) describes the structure (possibly dynamic) of the system, e.g. the ground in traffic simulations or the hyperlink network of a web-site [6]. Agents
are situated in this topology to define their neighborhood. Management of tag interactions by the environment is performed according to this topology. \textit{Rules} is the set of rules that define how the environment executes agent interactions.

Back to (8.4), $\Phi = \bigcup_i \phi_i$ is the set of references to softbodies in the system. The environment exploits softbodies to serve the population of agents (e.g. event notification to sentinels) and to enforce system rules by imposing environmental regulations. The regulation lets agents control their softbodies in a range of acceptable states defined by environment rules over $S_{\Phi}$. A softbody is consequently \textit{owned} and \textit{controlled} by an agent, while the control is \textit{regulated} by the environment.

We finally introduce \textit{TRANS}, a set of reaction rules $TRANS_{\Omega}$ and $TRANS_{\Phi}$ to model the regulated interaction mechanisms of the environment with agent softbodies. For any $\Phi$ and $\Omega$:

\begin{equation}
\begin{aligned}
\text{for } (\Omega, \Phi) \text{ and } \Omega \rightarrow \Omega', \exists A \subset \Phi: \forall a \in A, \exists (\phi_a, \phi'_a) \in S^2_{\Phi}: \phi_a \rightarrow \phi'_a \quad \text{TRANS}_{\Omega} \tag{8.6}
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
\text{for } (\Omega, \Phi) \text{ and } \forall a \in A, \phi_a \rightarrow \phi'_{\Omega} \quad \Omega \rightarrow \Omega' \text{ TRANS}_{\Phi} \tag{8.7}
\end{aligned}
\end{equation}

From an environment state $(\Omega, \Phi)$ and an evolution of the internals $\Omega$ to $\Omega'$, $TRANS_{\Omega}$ causes the set of softbodies $\Phi$ to evolve, so that for softbodies in the subset $A = \{a_1, \ldots, a_n\} \in \Phi$, the transformation $TRANS_{\Omega}$ entails all $\phi_i$ to evolve to some $\phi'_i$ for each $a_i$. Similarly, $TRANS_{\Phi}$ models the converse transformation. The subset $A$ depends on the \textit{Topology} and typically contains a ‘neighborhood’ of agents defined by application-dependent needs, such as an Euclidean or social (same taste, etc.) distance.

\subsection{8.1.2.3 Environment Mechanisms for Tag interaction}

This section describes the mechanisms of the environment in tag interactions.

\textbf{Agent Influence on the Environment.} When an agent intends to execute an action (i.e. change own public state or observe) in the BDI sense [98], its internals evolve from $\psi_{\text{init}}$ (intention selection) to $\psi_{\text{act}}$ (intention attempt). The agent is initially in a state $(\psi_{\text{init}}, \phi_{\text{init}})$, so that the change of internals causes the softbody to evolve due to the $INF_{\psi}$ reaction rule. The softbody consequently evolves to $\phi_{\text{act}}$:

\begin{equation}
\begin{aligned}
\text{for } (\psi_{\text{init}}, \phi_{\text{init}}) \text{ and } \psi_{\text{init}} \rightarrow \psi_{\text{act}}, \phi_{\text{init}} \rightarrow \phi_{\text{act}} \quad \text{INF}_{\psi} \tag{8.8}
\end{aligned}
\end{equation}

The modification of the softbody entails a reaction on the environment with the $TRANS_{\Phi}$ rule, from $\Omega_{\text{init}}$ to $\Omega_{\text{check}}$:

\begin{equation}
\begin{aligned}
\text{for } (\Omega_{\text{init}}, \Phi) \text{ and } \phi_{\text{init}} \rightarrow \phi_{\text{act}} \quad \Omega_{\text{init}} \rightarrow \Omega_{\text{check}} \text{ TRANS}_{\Phi} \tag{8.9}
\end{aligned}
\end{equation}
8.1. TAG INTERACTIONS AND SOFTBODY

The environment then checks whether its new state is valid according to applicable rules. If so, it continues the action by propagation of the effect to other agents (see sections 8.1.2.3 and 8.1.2.3), and it completes the process by informing the source agent. A successful action entails $\Omega_{ok}$. Success is observed with the softbody that becomes $\varphi_{ok}$ under the application of $\text{TRANS}_\Omega$.

$$\begin{align*}
\frac{\Omega_{\text{check}}, \Phi}{} & \text{ and } \Omega_{\text{check}} \rightarrow \Omega_{ok} \text{ TRANS}_\Omega \\
\varphi_{act} & \rightarrow \varphi_{ok} \\
(8.10)
\end{align*}$$

Fig. 8.3 shows the case where a public state is turned from ‘white’ to ‘black’ successfully.

Figure 8.3: Environment validates and commits the influence.

In case rules oppose the intention of the agent, the environment evolves from $\Omega_{\text{check}}$ to $\Omega_{nok}$ and counter-balances the agent attempt.

$$\begin{align*}
\frac{\Omega_{\text{check}}, \Phi}{} & \text{ and } \Omega_{\text{check}} \rightarrow \Omega_{nok} \text{ TRANS}_\Omega \\
\varphi_{act} & \rightarrow \varphi_{fail} \\
(8.11)
\end{align*}$$

Fig. 8.4 shows a case where the intention of the agent to turn its public state to black is not committed. Formula (8.11) follows formula (8.9) and cancels the action of the agent before its occurrence in the system (e.g. if one wants to push a wall, the body does not move). In the end, the softbody influences back the agent internals with $\text{INF}_\varphi$ to ‘report’ the opposition of the environment. In both cases, the execution follows the sequence $\text{INF}_\psi$, $\text{TRANS}_\varphi$, $\text{TRANS}_\Omega$, $\text{INF}_\varphi$.

Environmental Effect on Agents. The environment acts on agents with the mechanisms described in the previous section. However, the effect of the environment on the public state cannot be overruled by agents in the first place, so that the sequence of reaction rules differs. With the same notations as before:

$$\begin{align*}
\frac{\Omega_{\text{init}}, \Phi}{} & \text{ and } \Omega_{\text{init}} \rightarrow \Omega_{act} \text{ TRANS}_\Omega \\
\varphi_{\text{init}} & \rightarrow \varphi_{\text{update}} \\
(8.12)
\end{align*}$$

The environment acts on the softbody that consequently evolves to $\varphi_{\text{update}}$, representing either a change of public state (Fig. 8.5) or an observation received on
sensors. Then, the softbody informs its agent internals:

\[(\psi_{\text{init}}, \phi_{\text{init}}) \text{ and } \phi_{\text{init}} \rightarrow \phi_{\text{update}}, N \phi \]

In this rule, the agent cannot compensate yet the effect on the public state. Such a situation is shown on Fig. 8.5. The public state of the agent turns from white to black state under environmental effect and the agent internals are informed.

![Figure 8.5: Environmental effect on the agent public state.](image)

Agents can however react afterward in autonomy to either oppose or ignore the environmental effect. Typically, the agent can take subsequent actions to modify the environment, and such behaviors are governed by the same regulation sequence as in section 8.1.2.3. We can illustrate such situations with someone entering a river stream. The stream has an overwhelming strength at first and it carries the swimmer downstream (assuming the public state contains the position of the agent). In reply, the swimmer can try to oppose the stream and may succeed in crossing it if the swimming abilities and strength are sufficient.

**Public States Spread Management.** Given an agent, three types of events imply a spread of the public state in the environment, namely modification of the public state by the agent, environmental dynamics, and modification attempts on the agent public state by other agents through the environment. Fig. 8.6 illustrates the case where an agent modifies its public state (left part). Validation of the modification by the environment (central part) is followed by the publication of the resulting state to agents in the neighborhood defined by the topology (right part). We describe hereafter how to process the public state spread management in the three aforementioned cases.

**Agent modification.** Each agent controls its softbody and can modify the public state, under regulation by the environment. The procedure of modification is initially the same as the successful agent influence on the environment detailed in section 8.1.2.3. The sequential application of formulas (8.8) and (8.9) modifies the public state of the agent and validates it by the environment. Then, acknowledgment of the conformance of the new public state is performed with the publication
8.1. TAG INTERACTIONS AND SOFTBODY

Figure 8.6: Public State Management from left to right: the top-left agent modifies its public state; the environment validates the change; the change is spread to neighbors.

The neighborhood $A$ is included in the set of softbodies $\Phi \setminus \{a_{\varphi}\}$. Each softbody in $A$ receives a notification on its sensors about the change '$\varphi_{\text{act}} \rightarrow \varphi_{\text{ok}}$' of $a_{\varphi}$ (softbody of the agent that changed). Formula (8.14) is a generalization of formula (8.10) that only stated the successful completion of public state change without publication (case where the agent is alone in the system).

**Environmental dynamics** Environmental dynamics apply to subsets of agents. Typically, Archimedes' Law applies to agents under water in a simulation, while a clock interrupt to represent time concerns all agents in the system. The application of environmental dynamics on public state follows the procedure of section 8.1.2.3. Each dynamics corresponds to an environmental rule set that targets a particular type of public state variable. If $p$ in the public state of a softbody is the target of a rule, $p$ is assigned a new value $p'$ after application of the reaction operator $\text{TRANS}_\Omega$. Environment and softbody are consequently updated, and the new value $p'$ is spread in the system. The corresponding formula is a generalization of formula (8.12) that modifies all softbodies, similarly to (8.14).

**Attempts by other agents** When an agent intends to act on another agent to modify its public state (e.g. push an agent to change its position), the interaction is mediated by the environment. The procedure begins with a source agent that intends to act on the public state of a target agent. The intention leads to modifying the softbody with $\text{INF}_\psi$ and it entails an effect on the environment with $\text{TRANS}_\Phi$. 

\[
\frac{(\Omega_{\text{check}}, \Phi) \text{ and } \Omega_{\text{check}} \rightarrow \Omega_{\text{ok}}}{\varphi_{\text{act}} \rightarrow \varphi_{\text{ok}}, \exists A \subset \Phi \setminus \{a_{\varphi}\}, \forall a \in A, \varphi_0 \rightarrow \varphi_{\text{news}} \text{TRANS}_\Omega} \tag{8.14}
\]
If the action is authorized in the system, the environment reaction is three-fold by applying the action to the target agent, publishing the action to other agents, and sending an acknowledgment to the source agent:

\[
\Omega_{\text{check}}(\Phi) \text{ and } \Omega_{\text{check}} \rightarrow \Omega_{\text{ok}}
\]

\[
\phi_{\text{act}} \rightarrow \phi_{\text{ok}}^{\text{act}}, \phi' \rightarrow \phi_{\text{changed}}', \exists A \subset \Phi \setminus \{a_s, a_t\}, \forall a \in A, \phi^a \rightarrow \phi_{\text{news}}^a
\]

The neighborhood \( A \) is included in \( \Phi \setminus \{a_s, a_t\} \), where \( a_s \) is the softbody of the source agent and \( a_t \) the one of the target agent. In the end, each softbody informs its internals with \( \text{INF}_\phi \), so that agents are informed about the action. In particular, the target agent can react to this action.

### 8.2 Uni\( \phi \): Sentinels and autonomous agents

The model of tag interactions establishes a specific infrastructure that allows sentinels to fulfill their activities with preservation of autonomy. This infrastructure consists of a model of MAS with an explicit environment, agents endowed with a softbody, sentinels, and a specific presentation protocol between agents and environment to commit to the participation in tag interactions. Agents that choose not to complete the protocol decide autonomously to abandon the sentinel support provided in the system.

#### 8.2.1 Description of the MAS Uni\( \phi \) framework

Fig.8.7 depicts the model of MAS of the Uni\( \phi \) framework.

![Multi-Agent System](image)

**Figure 8.7: Model of Multi-Agent System**

MAS are defined as a set of application agents with softbody, sentinels that we can consider part of the host system, and an explicit environment. Application
agents fulfill the functional requirements of the system, while sentinels implement the quality requirements, relative to exception management in the system. The environment also fulfills the quality requirements in our approach, but its responsibilities and the underlying mechanisms could deal with functional ones as well.

The environment is presented as a centralized entity on the figure, but its implementation could be decentralized. This issue is further discussed in the last section of this chapter (page 134).

8.2.2 Environmental contract protocol

The infrastructure of MAS in the Uniφ framework encompasses a specific protocol for registration of agents in the system, named the ‘environmental contract protocol’. This protocol differs from ‘interaction protocols’ among agents as used throughout this document (although we use the same syntax), and merely means the procedure to register into the system.

The rationale of this protocol is to get the agreement and commitment of agents to participate in tag interactions for the sentinel support. Agents that register are supported by sentinels for system-level exception management. Others are not allowed to participate in the system activities in the case of the present protocol.

Fig. 8.8 presents the protocol. The essential functionality is for the environment to tell the agent the required information to expose on the public state, so that sentinel agents can access meaningful information relative to the activities they can support.

![Figure 8.8: Environmental contract protocol](image-url)
Agents introduce themselves with an enter call. The environment declares no-entry whenever new registrations are not allowed by the system owners. When registrations are possible, the environment sends a list of requirements that announces the information to expose on agent public states. If the agent cannot accept these requirements, it abandons the registration protocol. Otherwise, it complies and returns an accept message. The environment finally acknowledges the completion of the registration procedure.

Once an agent has registered, the environment can access its public state to execute and regulate the tag interaction mechanisms. Penalty could be applied to agents that do not comply with their commitment in the environment, but such additional mechanism is left for future extensions of the approach.

8.3 Discussion

The Uniφ framework allows sentinels to collect data about application agents without infringing their autonomy. It allows merging agent- and system-level exception management mechanisms with the aim to leveraging their respective advantages, while compensating their issues.

The framework adds however limited robustness to the system. Our focus was to preserve the autonomy, so that agents can still refuse to participate in exception handling procedures suggested by sentinels. Hägg assumed the agents always accept the participation, and the work of Klein et al. referred to the problem of unwilling agents in open systems [64]. Non-cooperative handling behaviors cause the sentinel to fail delivering their services, thus hampering the robustness acquired from system-level exception management mechanisms. The robustness to non-cooperative behaviors is managed by the agent execution model in a fundamental form, and more complex reactions require further research endeavors in this direction.

The environment is represented in the framework as a unique entity to which agents and sentinels refer. The environment can thus be implemented as a single centralized component of the MAS infrastructure. It can be implemented alternatively as a decentralized services, similarly to the distributed tuple-spaces services in Agilla and LIME [85, 1], or the centralized and decentralized action models for situated MAS available in the literature [26, 128]. Although decentralization is a desired property in MAS, the environment can be thought of as an abstraction of the system that can be implemented in centralized or decentralized fashion. The additional mechanisms required for decentralized control in the environment are considered out of the scope of this document.
The purpose of this chapter is to illustrate the Uniφ framework on a specific exceptional situation that can occur in MAS such as the case study, namely the agent death [64]. The sentinel approach was shown an appropriate solution to cope with the death of agents so as to maintain the activity of others.

9.1 Scenario of an agent death in the case study

In the Energy consortium, some agents are likely to disappear temporarily or permanently from the running system. Reasons are various, including a programming exception that exits the agent program, a restart of the hardware for maintenance, etc. If such event occurs during the enactment of a negotiation protocol, the agents running in the system must at least be able to maintain their activities and react correctly to the death of one of their peers, notably by deciding how to pursue the related activities and update their knowledge bases.

In the following, we extend the case study with a load-balancing mechanism. We suppose that each company of the consortium runs several agents that share the load of their business (these agents can run on different hardware), and one sentinel. When one of these agents fail, peer agents can take over their loads by two means.

Passive observation. Agents are required to expose two tags. Using these tags, agents can detect failures of their peers, assuming that a timeout means the death of a peer (agents are willing to work in this collaboration and we set aside other types of problems).

1. \texttt{Ref} is a reference to the goal the agent is attempting to achieve.
2. \texttt{ECD} (Expected Completion Date) indicates the time by which the agent declares to complete the goal.
Active observation. Sentinels observe tags in the system and alert running agents about timeouts if no one has already reacted to the corresponding death.

The environment of the system welcomes these worker agents and one sentinel per company. The death of one of the agents in a company requires others to reorganize. Sentinels are responsible to proactively inform agents about deaths in the company, so as to increase the recovery procedure.

9.2 Approach based on Autonomy-Ready Sentinels

Fig. 9.1 illustrates the scenario with two companies, represented by the two groups of agents $(S, a_1, a_2, a_3)$ and $(S', a_1', a_2', a_3')$.

Agents of the first group negotiate with agents in the second group, as defined in the Energy consortium. Sentinels observe actively agents in their respective groups and communicate with each other to inform about delays and recover time.

Exception management with passive observation. The execution procedure of Alg. 9.1 describes an overview of the agent execution, integrating the tag interaction mechanism to our execution model in the case of the particular consortium.

The execution cycle of an agent is represented as an infinite loop for compactness of the presentation (we do not show the event-based approach). The agent initializes an empty list of potential dead agents $\mathcal{X}_{\text{death}}$. The choice of the syntax $\mathcal{X}$ is deliberate, since this list would correspond to an expectation if the tag interaction was merged in the fundamental execution model as a standard mechanism.

The agent then uses the $\text{observe}()$ primitive that checks notifications received from the environment about the public states of peer agents. If the set of percepts contains an outdated $\text{ECD}$ tag, the corresponding agent identifier is added to $\mathcal{X}_{\text{death}}$. If this list is empty after analyzing all percepts, the execution continues with the


```plaintext
while true do
    Xdeath ← ∅;
    ecd_tags ← observe();
    foreach tag ∈ ecd_tags do
        if tag < tnow then
            Xdeath ← Xdeath∪{corresponding agent identifier};
        end
    end
    if Xdeath = ∅ then
        { No death: Standard execution }
        execute(S, t);
    else
        { At least one death: Prehend the reference to the task ref, and adopt it as a new goal }
        G ← G U {ref};
        execute(S, t);
    end
end
```

**Algorithm 9.1**: Execution procedure of application agents

*execute* procedure, relative to the case of agent deaths (the fundamental exception management mechanism is active, when the *execute* procedure is called). If the list is not empty, some agents have died; the agent prehends one goal from one of them, and executes it.

**Exception management with active observation.** The following procedure in Alg. 9.2 represents the execution sequence for sentinels. The role of sentinels focuses on improving the awareness of application agents by searching dead agents in the system to immediately notify running agents, which may have already detected the problem.

The procedure is made of two look-up loops on the list of agents in the system first to detect dead agents. The list of dead agents is then processed to inform running agents about potential death. The remarkable characteristic of this procedure is that sentinel cannot inspect the internal state of agents: They can only read public state information. Also, they do not take over control of agents, but simply inform them, so that they can leverage the extra information and react accordingly, and, most importantly, autonomously.

### 9.3 Execution example

In this example run, the agents execute a series of CNet, as defined in the Energy consortium. At some point in the execution, available agents a₁ and a₂ have to
while true do
    \{ Observation stage \} \( X_{\text{death}} \leftarrow \emptyset \);
    \foreach a \in A do
        if \( a.ecd < t_{\text{now}} \) then
            \( X_{\text{death}} \leftarrow X_{\text{death}} \cup \{ \text{corresponding agent identifier} \} \);  
        end
    end
    \{ Death management stage \} \foreach d \in X_{\text{death}} do
        \{ Inform running agents that some agents died if any \}
        candidate \leftarrow A \setminus X_{\text{death}};
        transfer(_, sentinel, candidate, inform, "d is dead");
    end
end

Algorithm 9.2: Sentinel execution procedure

fulfill the goals \((t_1, t_2, t_3)\). Fig. 9.2 represents the run where \(a_1\) dies and \(a_2\) exploits the tag information to recover the failure. Each interaction in this example follows the trace pattern \(INF_\psi, TRANS_\phi, TRANS_\Omega, INF_\phi\) presented in section 8.1.2.3 (influence of agents on the environment).

![Figure 9.2: Sequence of partial system states in the considered run (passive observation)](image)

Agent \(a_1\) starts executing to achieve \(t_1\), with a public state \((\text{Ref}, t_1)\) and \((\text{ECD}, d_1)\), where \(d_1\) is a date in the future. Agent \(a_2\) does the same and performs \(t_2\) with the corresponding public state where \(d_2 > d_1\). During the performance, \(a_1\) dies and \(a_2\) completes its current goal. Agent \(a_2\) starts a new cycle and receives information about its surrounding (passive observation with \text{observe}). As \(d_2 > d_1\), \(a_2\) can deduce that \(a_1\) is dead, so that it takes over the goal \(t_1\) to perform it instead of \(a_1\). In the last cycle, \(a_2\) observes no problem and performs the last goal \(t_3\). The work request is then completed, despite one death that has been automatically recovered.

Similarly, the same example run could rely on a sentinel to detect the death of \(a_1\). The difference is then that the sentinel detects the death and informs immedi-
ately $a_2$, so that $a_2$ can more rapidly engage in recovering the problem.

9.4 Discussion

Tag interactions and their supportive infrastructure allow sentinels performing their exception management tasks while preserving the application agent autonomy. The example run shows how the underlying mechanisms can be integrated in a system to leverage both agent- and system-level techniques.

The communication complexity of tag interactions is a potential limiting factor of the approach. The mechanism relies on event notifications from the environment for all changes on agent public states. In a topology where each agent can observe one another, the worse-case simultaneous notifications of changes on $n$ softbodies in the system would cost $O(n^2)$ messages, which is prohibitive, even with a decentralized environment. The environment features a configurable topology however, and can constrain agent neighborhoods with rules. For example, the designer can consider that agents are organized in a hierarchy, where the hierarchy is a binary tree. Assuming that agents can communicate with agents directly above and under the complexity remarkably drops to $O(1)$ (constants, either 2 or 3). In other words, an appropriate topology allows to control the communication complexity and should be analyzed against the application requirements.

The present state of the tag interaction model is however an extension of the fundamental agent execution model that have two shortcomings. First, the design of a system with tag interactions remains ad hoc, so that further endeavor should focus on a methodology to guide the choice of tags, matters such as topology, and functionalities of sentinels. In particular, a taxonomy of tags that lends themselves to typical exceptions could help in choosing or developing them. Second, tags allow detecting exceptional situations more efficiently by sentinels, but general exploitation mechanisms would be necessary to exploit this extra information. Reasoning models such as KGP or BDI could be configured to leverage this extra information.
Multi-agent systems are expected to feature many qualities in terms of flexibility and the capability to adapt automatically to the dynamics of their agents and 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 MAS [49, 64].

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 [120]— 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.
10.1 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 and operators, 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.

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 [64]. 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 [50, 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 [8]. 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 (pairs action-activation context), 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 merging of our agent-level execution model and corresponding architecture with the system-level sentinel approach into the Unip framework. The framework relies on our agent architecture with a ‘softbody’ and an active computational environment to introduce sentinels without infringing autonomy. Sentinels can advice agents to act in a particular way, but agents can decide autonomously whether to accept the advice: They are not controlled externally by sentinels.

10.2 Contributions to Agent-Based 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 non-deterministic 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 [41].

```java
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 [135]. 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.

10.3 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 [58, 117], 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, which yield a mean to create adaptive agents with regards to exceptional situations. 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.

10.4 Future perspectives

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 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


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


[52] Valérie Issarny. Concurrent Exception Handling. In Romanovsky et al. [104], pages 111–127. ix, 4, 28, 29


[65] Franziska Klügl, Manuel Fehler, and Rainer Herrler. About the Role of the Environment in Multi-agent Simulations. In Weyns et al. [130], pages 127–149. 126


[74] Marco Mamei and Franco Zambonelli. Motion Coordination in the Quake 3 Arena Environment: a Field-Based Approach. In Weyns et al. [130], pages 264–278. 126


[95] Eric Platon, Nicolas Sabouret, and Shinichi Honiden. Overhearing and direct interactions: Point of view of an active environment. In Weyns et al. [131], pages 121–138. 6


[121] Anand Tripathi and Robert Miller. Exception handling in agent-oriented systems. In Romanovsky et al. [104], pages 128–146. 24


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 [57, 19]. 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\(^1\) 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 [77] 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 [77].

---

\(^1\)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 pre-defined to reduce the complexity of the model, without impairing its semantics. A message is then defined as a record color set (keyword \texttt{colset}).

\begin{verbatim}
colset Msg = record from:STRING * to:STRING * content:STRING *
    sel:BOOL * sea:BOOL * eva:BOOL;
\end{verbatim}

The message is a record with similar field names as for the formal model. In addition, \texttt{sel} is a boolean that, if true, states that the message will have a handler available, none otherwise. Similarly, \texttt{sea} pre-define the success or failure for the handler search, and \texttt{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 [17].

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.

\begin{verbatim}
fun matchRel(r:RelevanceCriteria,p:Msg):BOOL=
    (* init *)
    if (#value r = init)
        then

\end{verbatim}
true
else
(* if there is any match, it is relevant *)
if(#value r = #dest p orelse #value r = #content p)
then
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 CPN Tools 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
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.
Publications

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


Résumé court de la thèse

Les systèmes multiagents (SMA) constituent une approche de conception logicielle dont les propriétés d'ouverture, d'hétérogénéité et d'autonomie sont reconnues comme particulièrement adaptées à la conception de systèmes complexes et distribués. Ces propriétés posent cependant des problèmes de robustesse. Le traitement des exceptions est une technique connue pour sa généralité et sa simplicité dans les problématiques de robustesse de fonctionnement, et son intégration dans les SMA est désirée. Les techniques traditionnelles ne sont cependant pas adaptées aux propriétés attendues.

L'objectif de ce travail est de présenter la notion d'exception de fonctionnement dans les SMA et de proposer un cadre de conception adapté. L'approche développée repose sur un modèle de traitement des exceptions qui garantit les propriétés des SMA: un agent peut décider seul (ouverture, hétérogénéité) si un événement doit être considéré comme une exception d'après des prédictions générées automatiquement et exactement sur les futurs événements attendus (autonomie). Ce modèle est décrit formellement et accompagné d'une architecture logicielle correspondante pour faciliter ses implémentations. L'approche est appliquée à un système complet afin de la comparer à des travaux existants et d'évaluer son coût d'exécution.

Mots-clés: Informatique, Intelligence artificielle, systèmes multiagents, conception orientée agents, traitement des exceptions, systèmes auto-adaptatifs, agents autonomes, robustesse de fonctionnement.

Short overview of the thesis

Multi-Agent Systems (MAS) are seen as an appropriate approach to deal with the growing complexity and distribution of modern software, owing to their properties of openness, heterogeneity, and autonomy. MAS properties entail however potential issues of robustness. Exception handling techniques were proposed to cope with such issues, with elegant and powerful mechanisms that should be introduced in MAS. Existing mechanisms are however insufficient to fully and correctly address the properties of MAS.

The purpose of this thesis is to present the notion of agent exception and to introduce an appropriate design framework. Our approach relies on an exception management model that guarantees the properties of MAS. Agents can decide individually (openness, heterogeneity) whether an event should be considered as exceptional, depending on expectations generated automatically and exactly relative to awaited events by the agent (autonomy). The model is formally described and interpreted into a software architecture that accommodates implementation endeavors. The approach is applied to a complete system, in order to compare the mechanism to existing ones and evaluate its computational cost.

Keywords: Computer Science, Artificial Intelligence, Multi-Agent Systems, Agent-Oriented Design, Exception Handling, Self-Adaptive Systems, Autonomous Agents, Robustness.