Università degli Studi di Torino

Computer Science Department

Ph.D. Program in Computer Science

Cycle XXXIII

Doctoral Thesis

Exception Handling for Robust Multi-Agent Systems

Stefano Tedeschi

Supervisor Prof. Matteo Baldoni

Ph.D. Program Coordinator Prof. Marco Grangetto

Revised version

October 15th, 2021

INF/01 - Computer Sciences

Stefano Tedeschi

Exception Handling for Robust Multi-Agent Systems

Supervisor: Prof. Matteo Baldoni

Ph.D. Program Coordinator: Prof. Marco Grangetto

Università degli Studi di Torino

Computer Science Department

Ph.D. Program in Computer Science

Cycle XXXIII

Corso Svizzera 185

10149 Torino, Italy

Abstract

Robustness is an important property of software systems, and the availability of proper feedback is seen as crucial to obtain it, especially in the case of systems of distributed and interconnected components. Exception handling has been successfully proposed in the past years as a powerful yet simple software engineering technology to address robustness in programming languages.

Multi-agent Systems (MAS), in turn, offer powerful abstractions for conceptualizing and implementing distributed systems, but the current design methodologies for MAS fall short in addressing robustness in a systematic way at design time. Thus, exception handling is usually approached by *ad hoc* solutions, that hamper code modularity and decoupling.

In this thesis we outline a vision of how robustness in MAS can be granted as a design property. We present a general model for multi-agent organizations that explicitly encompasses the notion of exception as a first-class element in the design of an organization. Relying on such a model, we propose an exception handling mechanism that is seamlessly integrated with organizational concepts, such as responsibilities, goals and norms.

The proposal is grounded on the notion of responsibility. In an organization, besides responsibilities for organizational tasks, we propose to specify also tasks and responsibilities for managing exceptions, that is, for providing feedback about the context in which exceptions occur, and for handling it. Agents will take on these responsibilities as soon as they take part in the organization.

We exemplify our vision on the JaCaMo multi-agent platform, by showing how its conceptual model and infrastructure can be extended so as to encompass the proposed exception handling mechanism.

Ai nonni.

Acknowledgements

The writing of this thesis has been sometimes quite hard and the results presented here would have never been possible without the support of many people.

First of all, I am heartedly thankful to Prof. Matteo Baldoni to whom I want to express my sincere gratitude and admiration. He not only constantly and rigorously guided me along my scientific path, but with his patience, friendly advices and passion, he truly transmitted to me the enthusiasm for doing research.

I owe a great debt to Prof. Cristina Baroglio and Dr. Roberto Micalizio, as well. The stimulating discussions we had throughout these years and their continuous help have been source of great inspiration and made me learn a lot.

I warmly thank the reviewers, Prof. Stefania Costantini (University of L'Aquila, Italy), Prof. Jomi F. Hübner (Federal University of Santa Catarina, Brazil) and Dr. Luca Sabatucci (ICAR CNR, Italy) for their time, suggestions and precious feedback.

The final part of of this research project has been carried out thanks to the grant "Bando Talenti della Società Civile" promoted by Fondazione CRT with Fondazione Giovanni Goria, which deserve an important acknowledgment.

I am grateful to Prof. Olivier Boissier, for having shared with me, through many inspiring and fruitful discussions, his deep knowledge about multi-agent systems and organizations.

I would like to thank all the colleagues – and friends – who contributed to making the time spent at the department so pleasant, in particular: Noemi, Francesco, Gianluca, Simone, Alessandra, Komal, and Livio.

A personal thank goes to my family – my mom, my dad, Arianna and my grandparents – who always and unconditionally supported and keep supporting me in whatever I attempt to pursue.

Finally, but importantly, a sweet acknowledgement to Milena, who has been constantly with me throughout this intense journey. She always encouraged me in the difficult moments and shared with me the joy of the happy ones, with love.

Thank you.

Torino, October 2021

Stefano

Publications

The candidate contributed to the following publications.

2021

Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., and **Tedeschi**, **S.** (2021a). "Demonstrating Exception Handling in JaCaMo". In: *Advances in Practical Applications of Agents, Multi-Agent Systems, and Social Good. The PAAMS Collection - 19th International Conference, PAAMS 2021, Salamanca, Spain, October 6–8, 2021, Proceedings*. Ed. by F. Dignum, J. M. Corchado, and F. De La Prieta. Vol. 12946. Lecture Notes in Computer Science. Springer, pp. 341–345. URL: https://doi.org/10.1007/978-3-030-85739-4_28.

Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., and **Tedeschi**, **S.** (2021b). "Distributing Responsibilities for Exception Handling in JaCaMo". In: *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '21. Virtual Event, United Kingdom: International Foundation for Autonomous Agents and Multiagent Systems, pp. 1752–1754. URL: http://www.ifaamas.org/Proceedings/aamas2021/pdfs/p1752.pdf.

Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., and **Tedeschi**, **S.** (2021c). "Exception Handling in Multiagent Organizations: Playing with JaCaMo". In: *Pre-Proceedings of the 9th International Workshop on Engineering Multi-Agent Systems, EMAS 2021, held in conjuction with AAMAS 2021*.

Baldoni, M., Baroglio, C., Micalizio, R., and **Tedeschi**, **S.** (2021d). "Reimagining Robust Distributed Systems through Accountable MAS". In: *IEEE Internet Computing*. To appear.

Baldoni, M., Baroglio, C., Micalizio, R., and **Tedeschi**, **S.** (2021e). "Robustness Based on Accountability in Multiagent Organizations". In: *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '21. Virtual Event, United Kingdom: International Foundation for Autonomous Agents and Multiagent Systems, pp. 142–150. URL: http://www.ifaamas.org/Proceedings/aamas2021/pdfs/p142.pdf.

Baldoni, M., Baroglio, C., Micalizio, R., and **Tedeschi**, **S.** (2021f). "Social Commitments for Engineering Interaction in Distributed Systems". In: *Artificial Intelligence Methods for Software Engineering*. Ed. by M. Kalech, R. Abreu, and M. Last. World Scientific. Chap. 3, pp. 51–85. URL: https://doi.org/10.1142/9789811239922_0003.

2020

Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., and **Tedeschi**, **S.** (2020a). "Accountability and Responsibility in Multiagent Organizations for Engineering Business Processes". In: *Engineering Multi-Agent Systems, 7th International Workshop, EMAS 2019, Montreal, QC, Canada, May 13–14, 2019, Revised Selected Papers*. Ed. by L. A. Dennis, R. H. Bordini, and Y. Lespérance. Vol. 12058. Lecture Notes in Computer Science. Springer, pp. 3–24. URL: https://doi.org/10.1007/978-3-030-51417-4_1.

Baldoni, M., Baroglio, C., Micalizio, R., and **Tedeschi**, **S.** (2020b). "Is Explanation the Real Key Factor for Innovation?" In: *Proceedings of the Italian Workshop on Explainable Artificial Intelligence co-located with 19th International Conference of the Italian Association for Artificial Intelligence, XAI.it@AIxIA 2020, Online Event, November 25-26, 2020*. Ed. by C. Musto, D. Magazzeni, S. Ruggieri, and G. Semeraro. Vol. 2742. CEUR Workshop Proceedings. CEUR-WS.org, pp. 87–95. URL: http://ceur-ws.org/Vol-2742/short2.pdf.

Baldoni, M., Baroglio, C., Micalizio, R., and **Tedeschi**, **S.** (2020c). "JADE/JaCaMo+2COMM: Programming Agent Interactions". In: *Advances in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness. The PAAMS Collection - 18th International Conference, PAAMS 2020, L'Aquila, Italy, October 7-9, 2020, Proceedings*. Ed. by Y. Demazeau, T. Holvoet, J. M. Corchado, and S. Costantini. Vol. 12092. Lecture Notes in Computer Science. Springer, pp. 388–391. URL: https://doi.org/10.1007/978-3-030-49778-1_33.

Tedeschi, **S.** (2020). "Engineering Multiagent Organizations Through Accountability". In: *Ambient Intelligence – Software and Applications. 11th International Symposium on Ambient Intelligence*. Ed. by P. Novais, G. Vercelli, J. L. Larriba-Pey, F. Herrera, and P. Chamoso. Vol. 1239. Advances in Intelligent Systems and Computing. Springer, pp. 305–308. URL: https://doi.org/10.1007/978-3-030-58356-9_36.

2019

Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., and **Tedeschi**, **S.** (2019a). "Accountability and Agents for Engineering Business Processes". In: *Pre-proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS 2019) held in conjuction with AAMAS 2019, Montreal, Canada, May 13-14, 2019*. Ed. by R. H. Bordini, L. A. Dennis, and Y. Lesperance. **Best Paper Award**. URL: https://cgi.csc.liv.ac.uk/~lad/emas2019/accepted/EMAS2019_paper_26.pdf.

Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., and **Tedeschi**, **S.** (2019b). "Engineering Business Processes through Accountability and Agents". In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019*. Ed. by E. Elkind, M. Veloso, N. Agmon, and M. E. Taylor. International Foundation for Autonomous Agents and Multiagent Systems, pp. 1796–1798. URL: http://dl.acm.org/citation.cfm?id=3331922.

Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., and **Tedeschi**, **S.** (2019c). "Engineering Multiagent Organizations by Accountability and Responsibility". In: *Discussion and Doctoral Consortium papers of AI*IA 2019 - 18th International Conference of the Italian Association for Artificial Intelligence, Rende, Italy, November 19-22, 2019*. Ed. by M. Alviano, G. Greco, M. Maratea, and F. Scarcello. Vol. 2495. CEUR Workshop Proceedings. CEUR-WS.org, pp. 12–23. URL: http://ceur-ws.org/Vol-2495/paper2.pdf.

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and **Tedeschi**, **S.** (2019d). "MOCA: An ORM model for computational accountability". In: *Intelligenza Artificiale* 13.1, pp. 5–20. URL: https://doi.org/10.3233/IA-180014.

Baldoni, M., Baroglio, C., Micalizio, R., and **Tedeschi**, **S.** (2019e). "Implementing Business Processes in JaCaMo+ by Exploiting Accountability and Responsibility". In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019*. Ed. by E. Elkind, M. Veloso, N. Agmon, and M. E. Taylor. International Foundation for Autonomous Agents and Multiagent Systems, pp. 2330–2332. URL: http://dl.acm.org/citation.cfm?id=3332102.

Baldoni, M., Baroglio, C., Micalizio, R., and **Tedeschi**, **S.** (2019f). "Programming Agents by Their Social Relationships: A Commitment-Based Approach". In: *Algorithms* 12.4, p. 76. URL: https://www.mdpi.com/1999-4893/12/4/76.

2018

Baldoni, M., Baroglio, C., Boissier, O., May, K. M., Micalizio, R., and **Tedeschi**, **S.** (2018a). "Accountability and Responsibility in Agents Organizations". In: *PRIMA 2018: Principles and Practice of Multi-Agent Systems - 21st International Conference, Tokyo, Japan, October 29 - November 2, 2018, Proceedings*. Ed. by T. Miller, N. Oren, Y. Sakurai, I. Noda, B. T. R. Savarimuthu, and T. C. Son. Lecture Notes in Computer Science 11224. Springer, pp. 403–419. URL: http://dx.doi.org/10.1007/978-3-030-03098-8_16.

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and **Tedeschi**, **S.** (2018b). "An Information Model for Computing Accountabilities". In: *AI*IA 2018 - Advances in Artificial Intelligence - XVIIth International Conference of the Italian Association for Artificial Intelligence, Trento, Italy, November 20-23, 2018, Proceedings*. Ed. by C. Ghidini, B. Magnini, A. Passerini, and P. Traverso. Vol. 11298. Lecture Notes in Computer Science. Springer, pp. 30–44. URL: https://doi.org/10.1007/978-3-030-03840-3_3.

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and **Tedeschi**, **S.** (2018c). "Computational Accountability in MAS Organizations with ADOPT". In: *Applied Sciences* 8.4. URL: http://www.mdpi.com/2076-3417/8/4/489.

Baldoni, M., Baroglio, C., Micalizio, R., and **Tedeschi**, **S.** (2018d). "Accountability and Responsibility in Business Processes via Agent Technology". In: *Proceedings of the Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion co-located with the Federated Logic Conference, RCRA@FLOC 2018, Oxford, United Kingdom, July 13, 2018*. Ed. by M. Maratea and M. Vallati. Vol. 2271. **Invited paper**. CEUR Workshop Proceedings. URL: http://ceur-ws.org/Vol-2271/invited1.pdf.

Dignum, V., Baldoni, M., Baroglio, C., Caon, M., Chatila, R., Dennis, L. A., Génova, G., Haim, G., Kließ, M. S., López-Sánchez, M., Micalizio, R., Pavón, J., Slavkovik, M., Smakman, M., Steenbergen, M. van, **Tedeschi**, **S.**, Torre, L. van der, Villata, S., and Wildt, T. de (2018). "Ethics by Design: Necessity or Curse?" In: *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society, AIES 2018, New Orleans, LA, USA, February 02-03, 2018*. Ed. by J. Furman, G. E. Marchant, H. Price, and F. Rossi. ACM, pp. 60–66. URL: https://doi.org/10.1145/3278721.3278745.

Tedeschi, **S.** (2018a). "Accountable Agents and Where to Find Them". In: *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society, AIES 2018, New Orleans, LA, USA, February 02-03, 2018*. Ed. by J. Furman, G. E. Marchant, H. Price, and F. Rossi. ACM, pp. 384–385. URL: https://doi.org/10.1145/3278721.3278783.

Tedeschi, **S.** (2018b). "Computational Accountability and Responsibility in the MAS Domain". In: *Proceedings of the AI*IA Doctoral Consortium (DC) co-located with the 17th Conference of the Italian Association for Artificial Intelligence (AI*IA 2018), Trento, Italy, November 20-23, 2018*. Ed. by M. Rospocher, L. Serafini, and S. Tonelli. Vol. 2249. **Honourable Mention**. CEUR Workshop Proceedings. URL: http://ceur-ws.org/Vol-2249/AIIA-DC2018_paper_5.pdf.

2017

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and **Tedeschi**, **S.** (2017a). "ADOPT JaCaMo: Account-ability-Driven Organization Programming Technique for JaCaMo". In: *PRIMA 2017: Principles and Practice of Multi-Agent Systems - 20th International Conference, Nice, France, October 30 - November 3, 2017, Proceedings*. Ed. by B. An, A. L. C. Bazzan, J. Leite, S. Villata, and L. W. N. van der Torre. Vol. 10621. Lecture Notes in Computer Science. Springer, pp. 295–312. URL: https://doi.org/10.1007/978-3-319-69131-2_18.

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and **Tedeschi**, **S.** (2017b). "Supporting Organizational Accountability Inside Multiagent Systems". In: *AI*IA 2017 Advances in Artificial Intelligence - XVIth International Conference of the Italian Association for Artificial Intelligence, Bari, Italy, November 14-17, 2017, Proceedings*. Ed. by F. Esposito, R. Basili, S. Ferilli, and F. A. Lisi. Vol. 10640. Lecture Notes in Computer Science. Springer, pp. 403–417. URL: https://doi.org/10.1007/978-3-319-70169-1_30.

2016

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and **Tedeschi**, **S.** (2016). "Computational Accountability". In: *Proceedings of the AI*IA Workshop on Deep Understanding and Reasoning: A Challenge for Next-generation Intelligent Agents 2016 co-located with 15th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2016), Genova, Italy, November 28th, 2016*. Ed. by F. Chesani, P. Mello, and M. Milano. Vol. 1802. CEUR Workshop Proceedings, pp. 56–62. URL: http://ceur-ws.org/Vol-1802/paper8.pdf.

Contents

1 Introduction

Contents

In the past years, software has acquired more and more a primary role in our daily lives. At the same time, software systems have become complex entities, where continuous interaction between a multitude of interconnected and heterogeneous components takes place in dynamic and distributed environments. Under this perspective, *robustness* is a fundamental requirement of such systems. A software is *robust* when it is able to keep an acceptable behavior in presence of abnormal execution conditions, like unavailability of system resources, communication failures, invalid or stressful inputs (Fernandez et al., 2005). The availability of proper *feedback* concerning the execution is seen as crucial to obtain it (Alderson and Doyle, 2010), especially in the case of systems of distributed and interconnected components. The problem of building robust software has implications on software engineering in general, and becomes more relevant due to the fact that we are witnessing an increasing spreading of "intelligent and autonomous" software in many aspects of our everyday life. Indeed, we already entrust software, sometimes even without noticing, with critical decisions in several scenarios. Modern planes, for instance, are equipped with (software) autopilots that not only help pilots in keeping the route, but even override the human pilots when their decisions might endanger the safety of the aircraft. It is therefore evident that, when intelligent software has the power to take decisions autonomously, we have a serious problem of making such software robust, that is, under given abnormal conditions it is able to determine what went wrong and take appropriate countermeasures.

One specific mechanism that supports robustness is *exception handling* (Goodenough, 1975a; Cristian, 1985; Buhr and Mok, 2000) which, roughly speaking, amounts to equipping the system with the capabilities needed to tackle classes of abnormal situations, identified at design time. An *exception* is typically conceived as an event that causes the suspension of the normal program execution, breaking the execution flow. Therefore, the purpose of an exception handling mechanism is to provide the tools to identify when an exception occurs and to apply suitable *handlers*, capable of treating the exception and recover. Programming languages research was among the first to address explicitly the concern of exception handling, with the aim of allowing programmers to build more reliable software. The main purpose of an exception handling system was to have systematic treatment of exceptional conditions at the language level to either recover an appropriate program state and resume the execution, or to terminate it 'gracefully', preventing the program from crashing. Throughout the years, exception handling proved to be effective in addressing robustness while also promoting software modularity and low coupling, and nowadays it is supported by most modern programming languages.

Studies in the field of Multi-Agent Systems (MAS) (Wooldridge, 2009), in turn, showed the effectiveness of an agent-oriented approach in modeling and implementing this kind of distributed, heterogeneous, and autonomous systems. The context gave rise to the development of many different programming paradigms and frameworks, such as, just to mention a few of them, JADE (Bellifemine et al., 1999), Jason (Bordini et al., 2007), and SARL (Rodriguez et al., 2014) for programming agents, and CArtAgO (Ricci et al., 2009) for programming agent environments. In other words, multi-agent systems are valuable for conceptualizing and implementing distributed systems. Nonetheless, surprisingly, most of the current design methodologies and platforms for their development fall short in addressing robustness in a systematic way, treating exceptions as part of their design. Thus, exception handling, despite few attempts, is usually approached by *ad hoc* solutions, with a negative impact on modularity and decoupling.

In this work we outline a vision of how robustness in multi-agent systems can be granted as a design property. We present an exception handling mechanism for use in MAS that is seamlessly integrated within some of the high-level concepts that characterize this paradigm; namely responsibilities, tasks and norms.

1.1 Motivation

Many modern systems "are complex networks of multiple algorithms, control loops, sensors, and human roles that interact over different time scales and changing conditions" (Woods, 2016). In sociology, such a complex network becomes a set of constraints that make a system, which comprises many parts, to act as a whole (Elder-Vass, 2011). The combination of individuals and relationships produces emergent powers that enable the system to achieve goals that otherwise would not be achievable (or not as easily). The same holds for multi-agent systems.

However, the greater complexity introduces also new fragilities, that need to be coped with. More generally, "... this complexity itself can be a source of new fragility, leading to 'robust yet fragile' tradeoffs in system design" (Alderson and Doyle, 2010). For example, consider an autonomous vehicle, as the one described in (Woods, 2016). It is equipped with eighteen sensor packages, basic sensor processing/actuator controls, reasoning software based on temporal logic, sensor fusion, multiple path, traffic, and mission planners, conflict management, health monitoring, fault management, optimization, classifiers, models of the environment (maps), obstacle detection, road finding, vehicle finding, and sensor validation checks. Here, the use of protocols, of layering, and of feedback creates a complex, multi-scale modularity that *per se* is exposed to many risks of failure in presence of abnormal conditions. If we think of such a complex system as a MAS, how to gain robustness?

Methodologies for MAS design and development typically assume that agents co-ordinate their interactions and tasks: system-level goals can be accomplished by taking advantage of the contribution of each agent (Timm et al., 2006). The agents'

autonomy, in turn, is an enabler of the system adaptability, which is crucial to achieve robustness: a robust system is one that adapts to stressful environmental conditions, and components can adapt to changing contextual conditions and perturbations only if they are autonomous in their decision process. Adaptability, however, requires the system to be equipped with the ability to produce proper feedback, and propagate it, so as to enable the selection and enactment of behavior that is appropriate to cope with the situation and recover. At the same time, in a distributed system, the agent detecting a perturbation may be not the one equipped with the means to address it. More importantly, the agent, which may have only a partial view of the system, could not be able to determine the impact of the specific perturbation over the overall distributed execution. A successful handling of the perturbation would then require the presence of a mechanism to make the detecting agents produce relevant feedback and to clearly identify the ones designated system-wide for processing such feedback for recovery. The lack of these mechanisms makes the system fragile.

Suppose, for instance, that an agent is requested to deliver a parcel, but the receiver's address is wrong. The parcel will not be delivered, but it is not the agent's fault nor the agent could solve the problem in isolation. As a result, the agent may be sanctioned but this would not help to achieve the result or to solve the problem. The system as a whole would have no information of the reasons of the failure. A proper feedback, delivered to the right agent, in turn, would allow to adapt to the situation and, for instance, gather the correct address for the subsequent delivery.

The problem is *the lack of, broadly speaking, (i) a feedback framework and (ii) a clear distribution of responsibilities among the agents concerning the handling of exceptional situations*. Such a lack, for instance, makes it impossible to acquire information about possible conflicts (that remain internal to the agents), and hinders the identification of other agents to which reassign a task because they have the skills that are needed to cope with a perturbation and compensate. As a consequence, the system will generally be unable of selecting alternative strategies for pursuing its goals in presence of unfavorable conditions.

To tackle these conditions effectively as a consequence of a good design, we need new conceptual and programming tools. Inspired by what has been proposed in the field of programming languages, we claim that *exception handling* can provide such a tool. As we will discuss in detail in the next chapters, exceptions allow a designer to distribute responsibilities among agents for raising and handling exceptions. It specifies how feedback concerning a given abnormal situation, that is collected by an agent, must be passed to another agent, who is in position to react to it. Thanks to such feedback, the latter agent, who in principle could even not know about the situation, becomes aware of the perturbation and is put in condition to trigger its internal deliberative process for deciding how to tackle it (with a straightforward benefit to the whole system). In our view, exception handling is, then, the key to design and develop robust multi-agent systems.

1.2 Research Objective

Given the importance of robustness in the design and development of software systems and the challenges arising from the peculiarities of an agent-oriented approach, as explained above, the main research objective of this manuscript is the following one:

> *To present an exception handling mechanism for use in multi-agent systems, encompassing exceptions as first-class elements, and based on the notions of responsibility and feedback.*

When agents join a MAS, they will be asked to explicitly take on the responsibilities: (i) for providing feedback about the context where they detected exceptions, while pursuing organizational goals, and (ii) if appointed, for handling such exceptions once the needed information is available.

To achieve the result, we rely on the *organization* metaphor. Multi-agent organizations (MAOs) (Corkill and Lesser, 1983; Zambonelli et al., 2003; Dignum et al., 2004a; Dignum et al., 2004b; Hübner et al., 2007; Fornara et al., 2008; Dastani

et al., 2009), indeed, are built upon the notion of responsibility (Vincent, 2011). For this reason, we believe that they are naturally suited to support the integration of an exception handling mechanism. An organization typically encompasses a decomposition of a global task into sub-tasks. Sub-tasks are, then, assigned to agents by means of *norms* (Boella et al., 2006; Boella et al., 2008; Singh, 2013), that orchestrate the execution: as soon as a specific organizational task is needed to be achieved, the normative system generates an *obligation* towards some agent to achieve that task. Agents' acceptance of the organizational constraints expressed by norms enables them to act in a shared environment, and achieve results unachievable if they acted in isolation.

An organization describes what is expected from the agents for achieving a global task, based on their supposed capabilities. However, agents may fail the expectations. When this happens, a normative system would typically react by issuing sanctions towards the misbehaving agent. Consequently, on one hand, we can see the normative system as a means that enables the orchestration of the activities of a group of autonomous agents, while on the other hand, in some sense we can see it also as a means that tries to produce robustness. This because the agents are pushed to do *what is expected of them*, and thus to tackle the situations the system is facing. The rationale is to guide the agents towards the interest of the organization. The problem is that sanctions are not generally accompanied by feedback and feedback handling mechanisms, and thus they do not provide the right means to supports robustness (Chopra and Singh, 2016; Baldoni et al., 2018b). To be effective, sanctions must at least (i) be sufficiently "strong" to contrast the agents' self-interest in pursuing different goals of their own, and (ii) target agents that actually have the resources and capabilities needed to face the situation of interest. In both cases robustness would be gained only by propagating through the system information about the *reasons* that caused the violation, and by revising the norms accordingly. Otherwise, for what concerns the first condition, how to identify a right trade-off that works for any agent without making assumptions of the agents' internals? For what concerns the second condition, how to propagate the reasons that caused the failure of some

agent? Finally, importantly, how to make such information reach the right agents, equipped with the capabilities (and willingness) needed to recover?

In essence, current organizational models leverage norms to shape the scope of the responsibilities that agents take when joining the organization, capturing what they should do to contribute to the achievement of the organizational goal (Sommerville, 2007; Sommerville et al., 2009; Feltus, 2014). We add that, in our view, responsibilities define the scope of the exceptions, expressed with respect to the organizational state, that agents ought to raise or handle, as well. In this work we show how an exception handling mechanism can be grafted on the normative system of multi-agent organizations, and its advantages in terms of increased robustness in the execution. We claim that the concept of responsibility not only allows modeling the duties of the agents in relation to the organizational goal, but that it also enables the realization of mechanisms for raising and handling exceptions that occur within the organization operation.

This is a pretty novel use of normative systems, which are traditionally used to support the realization of *correct* systems. Robustness and correctness are complementary concepts: while correctness is *"the ability of software products to perform their exact tasks, as defined by their specification."* (Meyer, 1988), robustness guarantees that if different cases do arise, the system will terminate its execution cleanly. We will show that, by introducing a proper infrastructure, both properties can be supported by the normative system, uniformly.

The proposed exception handling mechanism, despite being conceptually independent from any specific agent programming platform, will mainly set and evaluated within the context of the JaCaMo framework (Boissier et al., 2013), a well-known conceptual model and programming platform for multi-agent organizations. Its conceptual model and infrastructure have been extended in order to incorporate exception handling at the organizational level. In particular, we will discuss how to improve the specification of a JaCaMo organization by complementing the functional decomposition of the organizational goal with a set of *recovery strategies*. Such strategies realize the actual exception handling mechanism allowing agents to tackle

a given set of exceptional situations, which could occur during the achievement of organizational goals. Recovery strategies are then mapped to dedicated organizational goals, to be assigned to the agents when a specific perturbation is detected, in order to cope with it. Agents taking part in the organization are requested to explicitly take on the responsibilities for these goals.

Practical use cases will be presented, as well, in order to assess the benefits of the approach. Some of them take inspiration from the field of Business Process Management (Weske, 2007; Van der Aalst, 2013). Indeed, business processes realize a business goal by coordinating the tasks undertaken by multiple interacting parties. Under this perspective, when processes are by their nature distributed, multi-agent systems are good candidates to supply the right abstractions for realizing them.

The presented approach is strongly based on the notion of *responsibility* and inspired by the sibling notion of *accountability* (Garfinkel, 1967; Dubnick and Justice, 2004; Grant and Keohane, 2005; Baldoni et al., 2016; Baldoni et al., 2019). Indeed, the main aim of this Ph.D. has been to investigate the use of the two concepts to support the realization of robust multi-agent systems. At its core, accountability can be reduced to a key relationship between two parties: the former (the "account taker") can legitimately ask, under some agreed conditions, to the other an *account* about a process of interest; the latter (the "account giver") is legitimately required to provide such account to the account taker, if requested (Chopra and Singh, 2014). In many cultures, accountability is associated to liability and blame (Dubnick, 2013), but this view disregards the potential arising from the ability and designation to provide response about something to someone who is legitimated to ask. Accountability can be a useful tool for the realization of agent organizations that exhibit robustness as a design property. Indeed, accountability relationships realize channels through which feedback can be collected and propagated, so as to reach the right agents entitled to cope with it, in analogy to what happens with exceptions.

In the last part of the thesis, we will discuss a generalization of the proposed exception handling mechanism in terms of accountability. We will show how exception handling can be read, more generally, in respect of accountability relationships

among the agents participating in the organization. Accountability is an enabler for exception handling when the account about a perturbation (i.e., an exception) is given to the right agent, entitled to treat (i.e., handle) it and recover. Indeed, by way of accountability, an organization designer can specify how (relevant) contextual information produced during the achievement of goals flows from an agent to another through appropriate channels. The objective is to provide an adequate context for the account taker's decision-making, especially in front of invalid or exceptional situations.

It's worth noting that it is possible to interpret many system properties as types of robustness: reliability as robustness to component failures; efficiency as robustness to lack of resources; scalability as robustness to changes to the size and complexity of the system as a whole; modularity as robustness to structured component rearrangements; evolvability as robustness of lineages to changes on long time scales. We believe that the presented framework can be set the ground for capturing all these facets, as well as a wide range of non-functional requirements, besides robustness, such adaptability, explainability, reusability, and transparency. Our intuition is that these non-functional requirements are met in a distributed system when its components (agents in our perspective), can exchange contextual information at a different level of that of the outcomes that are specified by functional requirements. Accountability and responsibility can be valid conceptual tools for reaching this objective.

1.3 Thesis Outline

More in detail, the content of this manuscript is organized by following the structure below:

Chapter 2 introduces the challenges arising from the development of robust software by reviewing the literature on robustness in software systems, with particular attention to exception handling models adopted in programming languages and in distributed paradigms (such as the actor model).

Chapter 3 introduces multi-agent systems discussing their peculiarities and limitations w.r.t. robustness. The primary approaches to exception handling that have been proposed within the field are presented and discussed.

Chapter 4 presents an abstract model for exception handling in multi-agent systems, preserving the main features which characterize the agent paradigm; namely openness, heterogeneity, distribution, and autonomy. Exceptions are explicitly encompassed as a first-class element in the design of an agent organization and integrated with the high-level organizational concepts. An exception handling mechanism based on the presented model is illustrated. The mechanism is built on the idea of properly distributing responsibilities for raising and handling exceptions among the agents taking part in the organization.

Chapter 5 describes in detail the realization of an exception handling system based on the proposed model in the context of the JaCaMo framework for multi-agent organizations.

Chapter 6 evaluates the proposed exception handling mechanism by discussing a set of practical use cases. The main strengths and limitations of the approach are discussed and compared with the alternative proposals presented in the preceding chapters.

Chapter 7 discusses a generalization of the approach in terms of accountability by characterizing the concept for computational use. Exception handling is read back as a special case of accountability mechanism. Conclusions end the manuscript and introduce possible open directions for future work.

Exception Handling

2

Contents

The aim of this chapter is to introduce the main challenges related to the development of robust software systems. Exception handling is introduced as an effective tool to achieve robustness and the main features of some widely adopted exception handling models are discussed.

2.1 Robustness through Exception Handling

Robustness is an important property of software systems. The “ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary” 2010 defines it as:

The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

In many cases, robustness refers to a system property rather than to the system as a whole: a property of a system is robust if it is invariant with respect to a set of *perturbations* (Alderson and Doyle, 2010). This makes it possible to interpret many system properties as types of robustness: reliability as robustness to component failures; efficiency as robustness to lack of resources; scalability as robustness to changes to the size and complexity of the system as a whole; modularity as robustness to structured component rearrangements; evolvability as robustness of lineages to changes on long time scales.

In (Fernandez et al., 2005), robustness in software systems is defined as:

The ability of a software to keep an 'acceptable' behavior, expressed in terms of robustness requirements, in spite of exceptional or unforeseen execution conditions (such as the unavailability of system resources, communication failures, invalid or stressful inputs, etc.).

In other words, we can interpret robustness as the ability of a computer system to cope with abnormal or exceptional situations (also called *perturbations*) occurring during execution, which may cause the *failure* of some operations. For this to happen, its components must adapt their behavior to unexpected contextual conditions, showing some degree of autonomy in their decision process.

Let us consider a simple, but meaningful, example.

Example 1 (ATM). *Money withdrawal at an ATM involves two steps: (i) the user types the desired amount; (ii) the money is provided. Suppose the typed amount is read as a string (e.g., "100") and then parsed. If the string is not a number in digits (e.g., "one hundred") parsing fails. Prevention is impossible because only the parsing can tell if the data has the right format. The software realizing the ATM is robust when, instead of crashing, it copes with the perturbation, e.g., by asking the user again and, in case of repetitions of the mistake, by soliciting an operator.*

In general, building robust systems that encompass every point of possible failure is difficult. The availability of *feedback* concerning the perturbation is seen as crucial in gaining robustness (Alderson and Doyle, 2010), yet not easy to obtain as is the case of multi-scale systems or of distributed systems of interconnected components. We see feedback as a piece of information, broadly speaking some facts that are obtained retroactively, that objectively concern the execution of interest, and that are passed from one component to another so that they can be exploited for recovery. The significance and the quality of feedback are crucial, as well, in making a system robust: one would not want any kind of information to be returned but only information that is functional to the desired kind of robustness, and that comes from a reliable and informed source.

Among the different ways to achieve robustness, *exception handling* (Goodenough, 1975a; Cristian, 1985; Buhr and Mok, 2000) has been successfully proposed in the past years as a powerful yet simple software engineering technology. Exception handling is the process of responding to the occurrence of abnormal situations during the execution of a program. When an abnormal situation is detected (e.g. missing parameters, unknown format) an *exception* breaks the normal flow of execution and deviates it to a pre-registered *exception handler*, which is executed to handle the specific situation. On completion, the execution flow is then directed back to the program.

Recalling again the "ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary" 2010, an exception is defined as an:

> *Event that causes suspension of normal program execution.*

Generally speaking, *exception handling systems* (EHS) provide tools (language constructs, primitives, etc.) to (i) identify when abnormal events occur and (ii) treat them in order to recover. *Raising* (or *throwing*) an exception is then a way to signal that a given piece of the program could not execute normally - for example, when an input argument is invalid (e.g., the value is outside of the domain of a function) or when a resource it relies on is unavailable (like a missing file, a hard disk error,

or out-of-memory errors) - and that the situation requires a special handling. In this sense, we can conceive the exception as a feedback concerning the perturbation that is passed to the handler and whose informational content is to be exploited for recovery. *Handling* (or *catching*) the exception, in turn, refers to the set of instructions which have to be executed to restore the normal execution flow.

2.2 Exception Handling in Programming Languages

Programming languages research was among the first to address explicitly the concern of exception handling, with the aim of allowing programmers to build more reliable software, more easily. The first works entirely devoted to exception handling began to appear in the '70s (see (Goodenough, 1975a; Goodenough, 1975b; Goodenough, 1975c) for a set of initial proposals). The main purpose of an exception handling system was to have systematic treatment of exceptional conditions at the language level to either recover an appropriate program state and resume the execution, or to terminate it 'gracefully', i.e. ensuring the absence of any side-effect in stopping the execution (e.g., release reserved memory, persistent data consistency).

Following (Goodenough, 1975b), we have that:

> *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.*

From this seminal work, it clearly emerges that exceptions permit the user of an operation to extend the operation's domain (the set of inputs for which effects are defined) or its range (the effects obtained when certain inputs are processed). They allow tailoring an operation's results or effects to the purpose in using the operation, and they also allow generalizing operations, making them usable in a wider variety of contexts than would otherwise be the case.

Consequently, an exception's full significance is known only outside the detecting operation: the operation is not permitted to determine unilaterally what is to be done after an exception is raised. The invoker controls the response to the exception that is to be activated. This increases the generality of an operation because the appropriate "fixup" will vary from one use of the operation to the next. To this aim, an invoker must be given enough information about the failure.

This early definition of exception condition brings forward two invariant aspects, of primary importance for any exception handling mechanism.

1. Exception handling always involves two parties: a party that is *responsible* for raising an exception, and another party that is *responsible* for handling it.

2. Exception handling captures the need for some *feedback* (i.e., exactly the exception) that must be returned from the "invoked" to the "invoker" in order to cope with some situation of interest.

Providing the feedback amounts to raising an exception, whereas the response to such feedback amounts to handling it.

In summary, the author singles out the following specific properties of program exceptions:

1. An exception significance is known outside the detecting operation; thus, the point where an exception is raised is not also the point where it can be handled;

2. The invoker may be permitted to terminate the invoked operation;

3. The invoker controls whether or not a default response is to be activated.

Following the discussion in (Platon, 2007), most modern programming languages rely on a similar model of exception handling. When a program is in execution, the invocation of an operation can encounter an exceptional condition. An operation is any instruction or set of instructions that is called for execution. A characteristic of the exceptional condition is that, once such an event is detected in a block of instructions, the execution of such block cannot continue. The execution flow is

then deviated and a handler is searched so as to deal with the condition, until the execution of the program is resumed or terminated. The search is performed according to the current state of the program execution. Handlers are typically 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 the same one. If no handler is available where the exception has occurred, the handler search continues by examining 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 call stack is a record of the series of operation invocations that are done during the execution of the program. If no handler is available at the point where the exception is thrown, a handler is searched in the context of the previous "caller" in the stack. The search continues until a handler is found or when the call stack is entirely "unwound", which means the program cannot handle the exception at all and must terminate.

Moreover, an *exception model* defines the interactions between the syntactic unit in which the exception occurs (the thrower) and the handler. The *termination* model automatically terminates the thrower and destroys any objects within its scope. Once the exception is handled, control resumes at the next syntactic unit. In the *resumption* model, in turn, the computation continues from the point where the exception was originally thrown (Miller and Tripathi, 1997).

Many modern programming languages, such as Java, C++, C#, and Python, provide built-in support for exception handling following the termination model. In Java, for instance, the *try*, *catch* and *throw* keywords allow to define syntactic units where exceptions could be raised and where handlers are attached.

Let us recall Example 1. An excerpt of a possible Java implementation is reported in Listing 2.1, below.

```
1 public class ATMHandler {
2     private RequestHandler rh;
3     private MoneyKeeper mk;
4     public void withdraw() {
```

```
        try {
            int amount = rh.obtainAmount();
            mk.provideMoney(amount);
        }
        catch(AmountUnavailableException au) {
            ... operator ...
        }
    }
    ...
}

public class RequestHandler {
    private Reader r;
    private Parser p;
    public int obtainAmount() throws AmountUnavailableException {
        ...
        while(!done && count < 3) {
            amountString = r.getAmountAsString();
            try {
                amountInt = p.parseAmount(amountString);
                done = true;
            } catch(NotANumberException nan) {
                if(++count == 3) {
                    throw new AmountUnavailableException();
                }
            }
        }
        ...
    }
    ...
}

public class Reader {
    public String getAmountAsString() { ... }
    ...
}

public class Parser {
    public int parseAmount(String amountStr) throws NotANumberException {
        ...
        if(...) // String is not a number in digits
            throw new NotANumberException();
        ...
    }
    ...
}
```

Listing 2.1: Example of exception handling in Java.

Each class realizes a component of the ATM application. The `ATMHandler` realizes the top layer by providing a `withdraw()` method encompassing the whole withdrawal process. The amount of money is requested to the user by invoking the `obtainAmount()` method offered by the `RequestHandler` class (see Line 6) and then the money is provided through the invocation of the method `provideMoney(...)` of class `MoneyKeeper`. The `obtainAmount()` method (Lines 19-33), again, relies on the methods offered by two more classes: a `Reader` for obtaining the amount as a string and a `Parser` for parsing it. Let us focus on the latter one.

The `parseAmount(...)` method (Lines 43-48) declares, by means of the `throws` keyword, that an exception, called `NotANumberException`, could eventually be thrown during its execution and propagated to the calling method. Note that the `throws` keyword does not actually throw an exception. It specifies that an exception may occur in corresponding method. The `throw` keyword, in turn, is used to actually throw the exception within the method body.

To deal with this eventuality, in the caller method `obtainAmount()`, the keywords `try` and `catch` allow the programmer to define the handler and the syntactic unit where to attach it. The `try` block (Lines 23-26) specifies the set of instructions whose execution must be terminated, should the exception be thrown. The `catch` block, in turn, defines the set of instructions constituting the handler for the given exception. Note that multiple `catch` blocks (i.e., multiple handlers) could be defined to handle multiple exceptions and be attached to the same `try` block.

At runtime, should an exception occur upon invocation of `parseAmount(...)` (Line 24), the execution of the block would be terminated (and the following instruction would not be executed). The control flow would be then directed to the handler. Upon completion the normal execution flow would be resumed starting from the first instruction after the catch block.

The handler for `NotANumberException` could be also attached to other syntactic units of the program (e.g., at a higher level, in the `ATMHandler`'s `withdraw()` method). In that case, the exception would be propagated along the call stack, until the handler is found and applied.

As a final note, we highlight that exceptions can also be nested inside handlers. In other words, while executing a catch block to handle an exception, further exceptions could be thrown. This is what happens, for instance, at Lines 26-30.

2.2.1 Continuations

In functional programming, a *continuation* (Friedman et al., 1984; Reynolds, 1993) is an abstract representation of the control state of a program. In other words, a continuation reifies the program control state; it is a data structure that represents the computation at a given point in the process' execution. In languages that support continuations, the data structure can be accessed by the programming language, instead of being hidden in the runtime environment. Continuations are useful for encoding other control mechanisms in programming languages, such as coroutines, and notably exceptions. The term was firstly introduced by Adriaan van Wijngaarden in September 1964 (Wijngaarden, 1966).

First-class continuations are constructs that give a programming language the ability to save the execution state at any point and return to that point later in the program, possibly multiple times. When invoked, the current program state is replaced with the state at which the continuation was captured.

Interestingly, continuations allow a programmer to implement exception handling without having to rely on any dedicated mechanism built-in to the language[1]. As pointed out above, when an exception is thrown, control jumps back where the try block in which the exception occurred was created. The exception is then caught and the handler executed, if any. If no handler is available, control jumps back again along the call stack. Under this perspective, the call stack can be seen as a list of continuations.

With first-class continuations, we can capture the current continuation before evaluating a try block, then execute the block, and resume the previously saved continuation

[1] For a simple, yet clear, introduction to continuations and how to exploit them to deal with exception handling, check `https://archive.jlongster.com/Whats-in-a-Continuation`.

if any exception occur. This is possible because continuations can be resumed with argument values. If an exception is to be thrown within the block, the captured continuation will be called with an exception value enabling the execution of the appropriate handler.

2.3 Exception Handling in Distributed Systems

The exception handling model presented in the previous section is strongly based on the assumption that the execution is sequential and the program encompasses a single control flow. When an exception is thrown such a flow is deviated and the call stack is unwound until a suitable handler is found. However, such an assumption cannot be made in case of concurrent and distributed systems, where each component controls its own execution flow and no shared call stack can be unwound. For this reason, other mechanisms for exception handling have been proposed, such as *supervision* in the context of the actor model or *coordinated atomic actions* in distributed object systems.

2.3.1 The Actor Model and the Akka Framework

The notion of *actor* was originally introduced in 1973 in (Hewitt et al., 1973). Throughout the years, it has become an acknowledged theoretical basis for several practical implementations of concurrent, distributed and fault-tolerant systems, among which Akka[2] is currently one of the most established ones.

The actor model is a model of concurrent computation that conceives the actor as a universal primitive; all computational entities are modeled as independent actors that only communicate with others through message passing. In other words, there is not shared state among actors.

An actor is a computational entity which has a reactive nature. In response to a received message, it can concurrently:

[2] `https://akka.io/`

- Modify its own private state;
- Send a finite number of messages to other actors;
- Create a finite number of new actors;
- Designate the computation logic (behavior) to be used for the next message it receives.

Messages are passed between actors asynchronously and are processed by actors one at a time. Communication between the sender and receiver is then decoupled and asynchronous, allowing them to execute in different threads.

Akka is one of the most popular actor model frameworks that provide a complete toolkit and runtime for designing and building highly concurrent, distributed, and fault-tolerant, event-driven applications on top of the Java Virtual Machine.

The main constituents of an Akka actor are (Gupta, 2012):

State The actor objects hold instance variables that have certain state values or can be pure computational entities. These values define the state of the actor. The actor state can be changed only as a response to a message.

Behavior A behavior encodes the computation logic to be executed in response to a given message. Moreover, an actor can swap the existing behavior with a new one when a certain message arrives.

Mailbox The link between the sender and the receiver of a message is called mailbox. Every actor is attached to exactly one mailbox. When a message is sent to the actor, it is enqueued in the actor's mailbox, from where it will be then dequeued for processing by the receiving actor. It is possible to configure custom mailboxes; for instance a mailbox can be a simple message queue as well as a priority queue.

Lifecycle Every actor that is defined and created has an associated lifecycle. Broadly speaking, an actor lifecycle consists of three phases: (i) the actor is initialized and started, (ii) the actor receives and processes messages by executing specific

behaviors, and (iii) the actor stops itself when it receives a termination message. Akka provides some predefined hooks, which are triggered in some relevant states of the actor lifecycle, such as `preStart`, allowing the actor's state and behavior to be initialized, and `postStop`, to release any resource used by the actor.

Akka actors have each their own light-weight thread, which shields the actor's state from the rest of the system. In this sense, actors act as black-boxes to the rest of the system. The only way to indirectly access an Actor's state is through messaging. In Akka, each actor has associated a unique `ActorRef` which is the actor's address, or more precisely the address of its mailbox. The only way to send it a message is by knowing this reference.

The term *Actor System* is often used to refer to a collection of actors, their mailboxes, and their configuration.

2.3.2 Supervision in Akka

Akka actors are organized into a *supervision hierarchy*, which forms the basis of Akka's fault tolerance model. When actors are created, they are always created as children of another existing actor, which supervises them and manages their lifecycle. The creator actor becomes the *parent* of the newly created *child* actor. The underlying rationale is to break down the task to perform into smaller tasks to the point where it is simple enough to be performed by one single actor. As the complexity of the problem grows, the actor hierarchy also expands.

Actor hierarchies are represented as path structures. At the very top of the actor hierarchy is the root actor at `/`. There is then an actor called *guardian* whose path is `/user`. Any actor created within an actor system will be created as a child of the guardian or its descendants (e.g., `/user/myActor`).

Supervision is the basis of the "Let it Crash" fault tolerance model adopted by Akka. It can be conceived as a way to move the responsibility of responding to failure outside of the actor that can fail (Goodwin, 2015). Notably, this vision reflects

what initially postulated by Goodenough and explained in Section 2.2. Practically speaking, this means that an actor can have other child actors that it is responsible for supervising; it monitors the child actors (also called subordinates) for failures and can take actions regarding the child actor's lifecycle (e.g., restart it). Indeed, the parent having delegated the sub-tasks, is the one who can determine the impact of a specific failure of one of them onto the concurrent execution of the others.

Each time an actor faces a failure during the execution of a task, it can notify an exception to its parent actor, which, in turn, should implement suitable *supervision strategies* or escalate the exception to its parent again. The possible choices for the supervisor are the following ones:

Restart The supervisor will create a new actor and replace the old one (and all its descendants). The result is that the actor's internal state is reset. Two additional hooks, `preRestart()` and `postRestart()`, are available, allowing the actor to perform some operations respectively before and after being restarted by the supervisor;

Resume The failing actor will continue with the next message, discarding the message whose processing caused the failure;

Stop The supervisor will terminate the subordinate actor permanently;

Escalate The exception will be notified to the supervisor's parent which will be then in charge for treating it.

Moreover, Akka provides two modes for the supervision strategies, which can be adopted by actors:

One-For-One Strategy The supervision strategy is applied only to the failed child;

All-For-One Strategy The supervision strategy is applied to all the siblings of the failed child, as well.

Let us consider again Example 1 and illustrate how the scenario can be implemented in Akka[3].

```
class ATMHandler extends Actor {
    val rh = context.actorOf(Props[RequestHandler],"RequestHandler")
    val mk = context.actorOf(Props[MoneyKeeper],"MoneyKeeper")
    def receive = {
        case Withdraw => rh ! ObtainAmount
        case AmountObtained(amount) => mk ! ProvideMoney(amount)
        ...
    }
    override val supervisorStrategy = AllForOneStrategy() {
        case _: NotANumberException => Stop
    }
}

class RequestHandler extends Actor {
    var attempts = 0
    val r = context.actorOf(Props[Reader], "Reader")
    val p = context.actorOf(Props[Parser], "Parser")
    def receive = {
        case ObtainAmount | Restarted => r ! GetAmountAsString
        case AmountString(amountString) => p ! ParseAmount(amountString)
        case ParsingDone(amountInt) =>
            context.parent ! AmountObtained(amountInt)
    }
    override val supervisorStrategy = AllForOneStrategy() {
        case _: NotANumberException =>
            if(attempts < 2) {
                attempts += 1
                Restart
            }
            else {
                Escalate
            }
    }
}

class Reader extends Actor {
    def receive = {
        case GetAmountAsString =>
            ...
            context.parent ! AmountString(amountString)
    }
    override def postRestart(reason: Throwable): Unit = {
        super.postRestart(reason)
        context.parent ! Restarted
    }

```

[3]The Akka toolkit is available for both Java and Scala. The code illustrated in this work is written in Scala (`https://www.scala-lang.org/`).

```
47 }
48
49 class Parser extends Actor {
50     def receive = {
51         case ParseAmount(amountString) =>
52             ...
53             if(...) // String is not a number in digits
54                 throw new NotANumberException
55             ...
56     }
57 }
58
59 class MoneyKeeper extends Actor {
60     def receive = {
61         case ProvideMoney(amount) => ...
62     }
63 }
```

Listing 2.2: Example of exception handling in Akka.

In analogy with the Java implementation, we realize each component of the ATM system as a separate actor. Differently from Java, where the interaction among the objects (i.e., classes instances) occurred through method invocations, here all the interaction must be modeled in terms of messages to be exchanged.

The method `context.actorOf(...)` allows to create a new actor, as a child of the actor in which the method is invoked. In our case the `ATMHandler` actor will have two children (see Lines 2-3), as well as the `RequestHandler` (Lines 16-17).

Moreover, each actor is equipped with a `receive` method describing how the actor should react to the reception of a given set of messages. For instance, when receiving a `Withdraw` message, the `ATMHandler` will send an additional message to its child, the `RequestHandler`, asking it to obtain the amount (Line 5). The `!` operator denotes the sending of a message.

Should the amount string not be a number in digits, the `Parser` actor would throw a `NotANumberException` (Line 54). To deal with such a a circumstance, a suitable supervision strategy is defined in its parent actor, namely the `RequestHandler`. In particular, the parent restarts its children up to three times so that a new amount can be collected. Note that the `Restart` strategy is applied to all the children because

we have an `AllForOneStrategy`. After three attempts the exception is escalated to the `ATMHandler` which finally handles it by stopping all its children (Lines 9-11).

Exception handling in Akka borrows the strengths of the model presented in the previous section for sequential programs and adapts it to a distributed and concurrent context of the actor model. Indeed, in both approaches, a clear structure, defining how the information related to the failure (i.e. the exception) should be encoded and should flow, is defined. In sequential programs this structure amounts to the call stack whilst in Akka it follows the chain of parent-child relationships among actors.

2.3.3 Coordinated Atomic Actions

Complementary to the actor model, in the context of distributed object systems, the *Coordinated Atomic Action* (or *CA action*) concept (Xu et al., 1995; Randell et al., 1997; Xu et al., 1998; Xu et al., 2000; Romanovsky, 2001; Pereira and Melo, 2010) has been proposed as a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components. It provides a conceptual framework for dealing with different kinds of concurrency and achieving fault tolerance by extending and integrating two complementary concepts: *conversations* and *transactions*. Conversations are used to control cooperative concurrency and to implement coordinated error recovery, while transactions are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency.

A CA action performs a set of operations on a group of distributed objects atomically, and thus behaves like a transaction. However, the body of a CA action can be multi-threaded. In other words, CA actions also behave like conversations in that they allow a set of threads to come together in order to perform some action atomically.

The interface of a CA action specifies the objects that are to be manipulated by the CA action and the roles that can manipulate these objects. In order to perform a CA

action, a group of threads must come together and agree to perform each role in the CA action concurrently with one thread per role.

During the execution of a CA action, one of the threads that are involved in the action may raise an exception. If that exception cannot be dealt with locally by the thread, then it is propagated to the other threads involved in the CA action. Since it is possible for several threads to raise an exception at more or less the same time, a process of exception resolution (Campbell and Randell, 1986) is put in place to agree on the exception to be propagated and handled within the CA action. The notion of *exception tree* allows to impose a partial order on the exceptions that could be raised within a CA action so that a higher exception has a handler which is intended to handle any lower level exception.

Once an agreed exception has been propagated to all of the threads involved in the CA action, then some form of error recovery mechanism can be invoked. More precisely, every component of the atomic action responds to the raised exception by changing the normal control flow to an exceptional one which executes a handler for that exception. It may still be possible to complete the performance of the CA action successfully using *forward error recovery*. Conversely, it may be possible to use *backward error recovery* to undo the effects of the CA action and start again, perhaps using a different variant of each role. If it is not possible to achieve either a normal outcome or an exceptional outcome using these error recovery mechanisms, then the CA action is aborted and its effects undone.

2.4 Exception Handling in Business Process Management

In the context of Information Systems, Business Processes (BPs) are widely used to capture how a service or a product is brought forth by a set of combined activities, that may involve multiple, directly or indirectly, interacting parties. Weske, (2007) defines a business process as:

A set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal.

In general, a business goal is achieved by breaking it up into sub-goals, which are distributed to a number of actors (either human or software). Each actor carries out part of the process, and depends on the collaboration of others to perform its task. In this context, the ability to handle perturbations possibly occurring during the distributed execution is of uttermost importance. For this reason, modern formalisms and notations for modeling business processes encompass fairly sophisticated exception handling mechanisms.

Business Process Management (BPM) can be defined as the combination of information technology and management science to construct operational business processes (Van der Aalst, 2013). In other words, BPM provides formal and practical tools to encode a set of unstructured business activities in a structured business process, assuring a stable and maintainable organization of work. Software systems that supports BPM are called Business Process Management Systems (BPMS), and are used to design, enact and manage operational business processes.

Different notations were proposed as graphical languages to express process models, like UML (Unified Modeling Language), EPC (Event-driven Process Chain), or Petri-nets. Petri-nets were the first graphical formalism widely (and still currently) adopted for representing business processes. The reason can be found in the centrality of concurrency, that Petri-nets firstly manage through fork/join primitives. A notation that quickly became a de facto standard is BPMN (White, 2004), maintained by the Object Management Group. Its current version is the 2.0, released in 2012. The output of the process design with BPMN is a Business Process Diagram (BPD), a flowchart-based graph representing the desired sequence(s) of activities in order to obtain a specific result.

In the following we briefly introduce the BPMN notation along with its adopted formalism to represent exception handling during the execution of a BP.

2.4.1 BPMN Basics

A BPMN diagram has a set of three core elements, which are *flow objects* (Silver, 2011):

Activity An activity in BPMN is an action, a unit of work performed, with a well-defined start and end. It is represented by a rounded rectangle. Every activity is either a *task* or a *subprocess*. A task is atomic, meaning that it has no internal subparts described by the process model. A subprocess is compound, meaning that it has subparts defined in the model.

Event An event is represented by a circle and encodes "something that happens" during the course of a business process. Events usually have a cause (trigger) or an impact (result). More precisely, a BPMN event describes how the process generates or responds to a signal that something happened. Events are circles with open centers to allow internal markers to differentiate different triggers or results. There are three types of events, based on when they affect the flow: *start*, *intermediate*, and *end* events.

Gateway A gateway is represented by the diamond shape and it encodes a branch point in the process flow. Thus, it will determine traditional decisions, as well as the forking, merging, and joining of paths. Internal markers indicate the type of behavior control.

Flow objects are then connected together in a diagram to create the basic skeletal structure of the business process. In particular, the *sequence flow* is used to show the order in which activities have to be performed.

Figure 2.1 shows a very simple business process in BPMN. Once the process starts, activity `A` is the first to be executed. Once `A` has been completed, the parallel gateway denotes that activities `B` and `C` must be executed in parallel. When both activities are completed, activity `D` can be performed and, after that, the process ends.

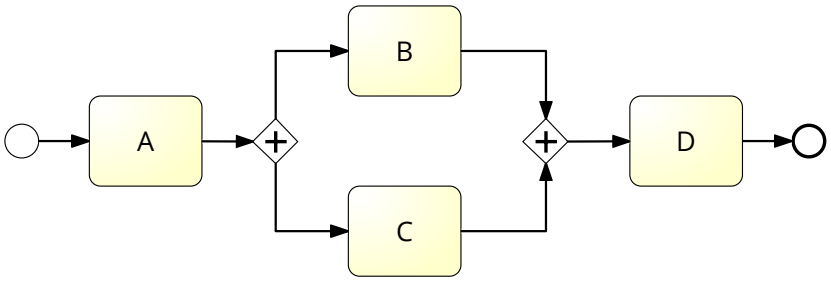

Figure 2.1: A very simple BPMN diagram.

2.4.2 Error Events

BPMN encompasses an effective mechanism for exception handling, built on top of the notion of *error event*. Error events aim at handling the occurrence of errors during the execution of a certain activity or at a certain point in the flow of a process. An error event, in fact, represents an exception end state of a process activity.

An error event on the boundary of a task (denoted in white) represents the fact that the task could end with an exception. The normal flow, the sequence flow out of the task, represents the exit when the task completes successfully, and the exception flow, the sequence flow out of the error event, is the exit when it does not.

Moreover, in expanded subprocesses error end events could also occur. Error end events (denoted in black) are used to indicate that a certain process path ends with an error. Such an event throws an error signal to the boundary of the subprocess, where it is propagated outside the subprocess.

Figure 2.2 shows the BPMN diagram of a business process realizing the ATM scenario of Example 1. The activity encoding the withdrawal can be expanded in a subprocess, namely encompassing two activities: `Obtain Amount` and `Provide Money`. `Obtain Amount` can be further expanded.

Let us focus on task `Parse Amount`. The task specifies a boundary error event, encoding that the task could end unsuccessfully with a `Not A Number` error. Should such a circumstance occur, the sequence flow of the process would go back to the `Count Attempts` task, at the beginning of the subprocess. The amount is

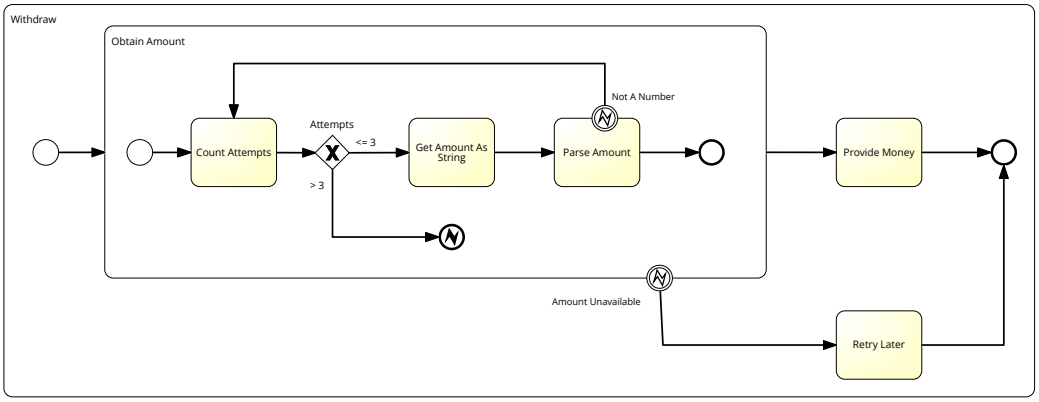

Figure 2.2: Example of exception handling in BPMN.

requested to the user up to three times. After three attempts have been performed unsuccessfully, an error end event is triggered and an `Amount Unavailable` error is propagated outside the subprocess. In the parent activity, the error is finally handled by a dedicated `Retry Later` activity.

As before, error events in BPMN allow to clearly separate normal flows from exceptional ones. At the same time, they allow to define suitable handlers, uniformly modeled as activities to be executed as soon as an error is signaled, in order to restore the normal execution flow in the process.

2.4.3 Event Subprocesses

BPMN 2.0 introduced a further construct to model the handling of specific events within a process: the *event subprocess*. An event subprocess is located within another process and is identified by a dotted-line frame. It is triggered by a single start event (e.g., an error event) and models the process to execute when the corresponding event occurs in the enclosing process. Start events can be interrupting or non-interrupting ones. Depending on the type of start event, the event subprocess will stop the enclosing subprocess, or it will be executed simultaneously.

Event supbrocesses are typically used to handle exceptions. Figure 2.3, in particular, shows the BPMN diagram of the ATM scenario, where the `Retry Later` activity is embedded in an event subprocess triggered by the `Amount Unavailable` error

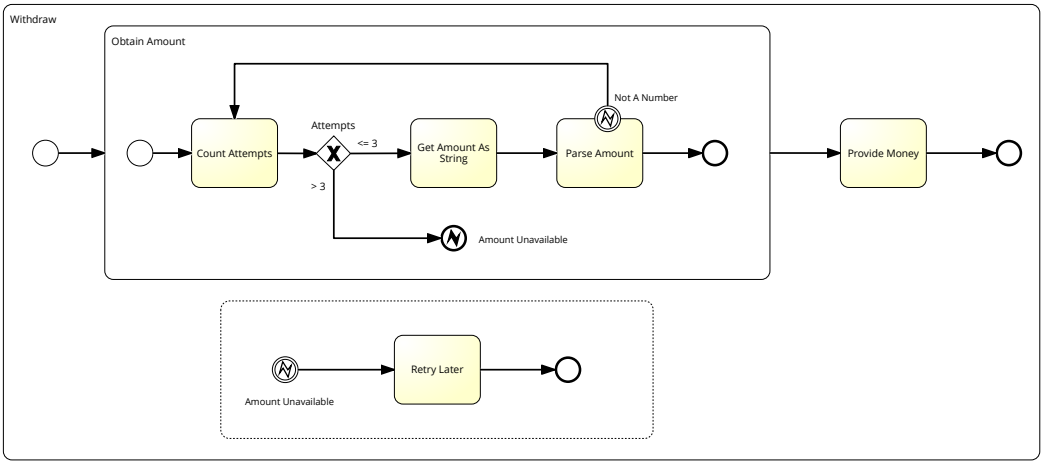

Figure 2.3: Example of exception handling in BPMN with event subprocess.

event. As soon as the error event is signaled, the execution of the `Withdraw` process is aborted and the subprocess is instead executed to handle the exception.

2.5 Exception Handling in Self-Adaptive Systems

The treatment of perturbations is of interest also in the perspective of self-adaptive systems. Self-adaptive software, in particular, is capable of evaluating and changing its behavior, whenever an evaluation shows that the system is not accomplishing what it was intended to do, or when better performance may be possible (Macías-Escrivá et al., 2013). In other words, a self-adaptive system has the ability to autonomously modify its behavior at runtime in response to changes in the environment (Cheng et al., 2009; Whittle et al., 2009; Lemos et al., 2013), such as the occurrence of some perturbation.

Common to most of the existing approaches is the identification of four processes which are fundamental for adaptivity: *monitoring*, *analyzing*, *planning*, and *executing* (IBM, 2005). They constitute a sort of closed-loop control over a target system (also known as *feedback control system* in control theory (Seborg et al., 2010)). These processes together allow to monitor the system, reflect on observations for problems, and control the system to maintain it within acceptable bounds of behavior (Garlan

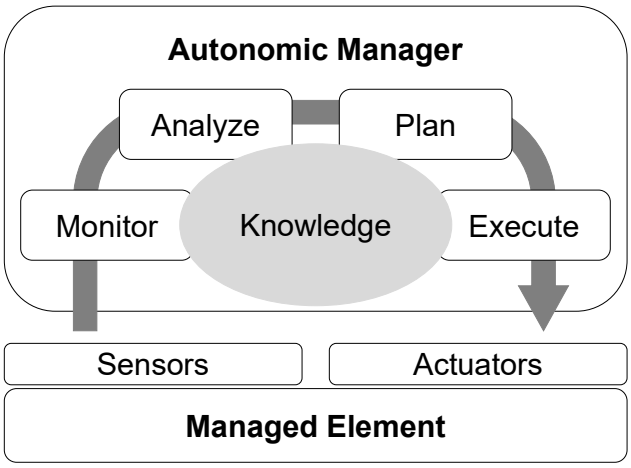

Figure 2.4: MAPE-K loop in the IBM autonomic framework (IBM, 2005).

et al., 2009). The loop is illustrated in Figure 2.4 and it typically encompasses two parts: an *autonomic manager* and a *managed element*, which could be a system or a component within a larger system. More precisely, each phase is characterized as follows (Garlan et al., 2009):

Monitoring concerns extracting relevant information out of the managed element;

Analyzing amounts to determining if some perturbation has occurred inside the system (e.g., because a system property exhibits an out-of-bound value);

Planning includes the mechanisms to select a course of action to adapt the managed element once a specific perturbation is detected;

Executing allows to schedule and perform the necessary planned changes to the system.

All the processes share a *knowledge component*, which contains the models, data, and facts needed by the autonomic manager to effectively put the adaptation process in place. For this reason, the loop is often called MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge).

As an illustration, the well-known Rainbow architecture (Garlan et al., 2009) implements the MAPE-K loop with the aim of facing unexpected events. It provides mechanisms to monitor a target system and its environment, detecting events that

denote opportunities for adaptation, selecting a course of action to address these opportunities, and apply changes. Once a problem is detected in the system (i.e., a violation of an architectural constraint), an *adaptation strategy* that suits the problem is selected and the framework coordinates the execution of that strategy. A specific language to express adaptation strategies allows one to specify for what to adapt, when to adapt, and how to adapt the system. Strategies are executed by a dedicated module in the architecture and capture a pattern of adaptation in which each step evaluates a set of condition-action pairs and executes an action over the system.

In (Sabatucci et al., 2018), the authors propose a unified metamodel for describing the various categories of self-adaptation approaches. In particular, four types of adaptive smart systems are identified, each one respectively exhibiting a greater degree of adaptation capacity. A system belongs to the first type if it owns an entity composed of the elements necessary to know, at design time, the environment and to act on it. The second type additionally requires an engine that is qualified to know the runtime model of the system, in order to possibly change or influence the solution strategy built at design time. The third type involves a more complex element, able to build completely new solution strategies through a repository of established functionalities. Finally, the fourth type requires the presence of an engine able to set up additional solution builders along with the evolution of the system.

It's worth noting that multi-agent systems seem particularly suitable to realize self-adaptive systems, as advocated in (Weyns and Georgeff, 2009; Macías-Escrivá et al., 2013; Krupitzer et al., 2015). Indeed, the goal oriented-behavior, loose coupling, and context sensitivity of agents provides the flexibility which is needed for self-adaptivity and reuse. Nonetheless, a recurring open challenge in both fields is related to the notion *unexpected exception*, i.e., a deviation from the expected behavior of the system that is not explicitly defined "by contract". These kinds of exceptions are particularly difficult to tackle because the way in which they ought to be handled, and the associated responsibility, cannot be anticipated. Instead, they are emergent and thus suitable handlers must be built from scratch as soon as the specific perturbation occurs, at runtime.

3 Exception Handling in Multi-Agent Systems Literature

Contents

To be suitable for MAS, an exception handling mechanism should leverage on both the proactivity of agents and the environment in which agents are situated. Throughout the years, different approaches have been proposed in the context of the multi-agent systems literature. Nonetheless, no clear consensus has been reached within the community.

One main difficulty arising in the development of an exception handling system for MAS is to conjugate an effective model for exception detection and recovery with the peculiarities characterizing the agent paradigm, i.e., autonomy, heterogeneity, openness, distribution, and situatedness.

In this chapter, after a brief introduction to the basics of multi-agent systems, we review the most prominent approaches that have been proposed to address the problem of building robust and fault-tolerant MAS.

3.1 Background on Multi-Agent Systems

In the field of Artificial Intelligence (AI), a Multi-Agent System (MAS) is a computerized system composed of multiple interacting computing elements (Wooldridge, 2009), called *agents*, within a shared and possibly distributed environment. In these kinds of systems, agents represent several autonomous components which use common resources and interact to achieve individual or shared goals. The motivation for studying MAS often stems from the fact that they allow to model complex, heterogeneous, distributed and dynamic systems in a straightforward way.

Agents are characterized by some important features. First, they are *autonomous*, i.e., capable (to some extent) to independently decide and take action in order to satisfy their design objectives (Wooldridge, 2009). Second, they are capable of interacting with others, engaging social relationship with other agents. Another important characteristic of agents is their *situatedness*; agents do not exist on their own, but inside an environment (either physical or virtual) that they can perceive through *sensors* and upon which they can act through *actuators* (Russell and Norvig, 2002; Bordini et al., 2007; Wooldridge, 2009).

Different definitions of what an agent is have been proposed and there is no universally accepted definition of the term. However, some main properties of agents can be identified (Wooldridge and Jennings, 1995; Wooldridge, 2009):

Autonomy The ability to operate without external intervention, having control over their internal states and actions, in order to decide how to act so as to accomplish some delegated goals;

Social ability The ability to interact with others to satisfy their goals;

Reactivity The ability to perceive the environment and timely respond to changes occurring in it;

Proactiveness The ability to exhibit a goal-oriented behavior that is not only driven by external events, but involves taking the initiative to satisfy their design objectives.

Autonomy ensures that agents own control of their execution flow and private data. The notion of autonomy is related to the one of *encapsulation* in object-oriented systems. One big advantage of conceiving agents as autonomous entities, is that autonomy increases modularity and decoupling.

Russell and Norvig, (2002) point out the importance of *rationality* in an agent's behavior. An agent is rational if it is able to select the action which maximizes a given *performance measure*, according to the evidence provided by its percepts and its internal knowledge.

3.1.1 BDI Agents

Agents are often modeled by leveraging high-level abstractions inspired from intentional notions of the human behavior, borrowed from philosophy. The *belief-desire-intention* (BDI) model (Bratman, 1987; Bratman et al., 1988) attributes to agents some *mental states*, such as beliefs, desires, and intentions. Following (Bordini et al., 2007), the three concepts are characterized as follows:

Beliefs represent the agent's knowledge about the world;

Desires represent, roughly speaking, the goals delegated to the agent, i.e., the states of affairs that the agent might want to accomplish;

Intentions represent states of affairs the agent is actively pursuing.

The agent execution model is then conceived in terms of these mentalist notions (Shoham, 1993); actions executed by an agent are the result of a deliberation and reasoning process consisting of the selection of what intentions to pursue among the possible options (i.e., the agent's desires) and according to the agent's current beliefs.

Nowadays, the BDI model has been widely accepted in the MAS community as an effective theoretical framework for modeling agents. A wide number of architectures, such as PRS (Procedural Reasoning System) (Georgeff and Lansky, 1987), and programming languages have been proposed following such model. Some examples include AGENT-0 (Shoham, 1993), 3APL (Hindriks et al., 1999), GOAL (Boer et al., 2007), 2APL (Dastani, 2008), and the well-known AgentSpeak(L) (Rao, 1996), whose most established implementations are Jason (Bordini et al., 2007) and ASTRA (Collier et al., 2015).

Autonomy, social ability and high level of abstractions make agents particularly suitable to build concurrent, distributed and heterogeneous systems. *Heterogeneity* means that agents in a multi-agent system could be developed by different stakeholders using different technologies, architectures and programming languages. However, heterogeneity makes interoperability more difficult. This issue is strictly related with the notion of *openness*. Open MAS are those systems where agents can enter and leave at any time, dynamically.

As we will se in the upcoming chapters, these features, despite constituting the core of the agent paradigm, pose some significant challenges w.r.t. the realization of an effective exception handling mechanism.

3.2 The Guardian

As a first attempt to introduce exception handling in MAS, let us consider the *guardian*. It is a model for exception handling in distributed and concurrent systems,

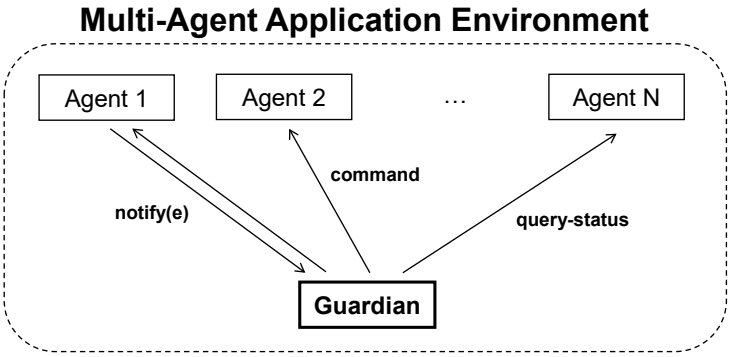

Figure 3.1: Multi-agent application environment in the Guardian model, from (Tripathi and Miller, 2001).

originally presented in (Tripathi and Miller, 2001; Miller and Tripathi, 2004), which realizes coordinated exception handling. The proposal was originally developed in the context of distributed-object systems, and then mapped to a multi-agent context. This model addresses two fundamental problems with distributed exception handling in a group of asynchronous processes. The first is to perform recovery when multiple exceptions are concurrently signaled. The second is to determine the correct context in which a process should execute its exception handling actions.

The guardian in a distributed program represents a global exception handler, which encapsulates rules for handling concurrent exceptions and directing each process to the correct context for executing recovery actions. More in detail, the guardian is a distributed global entity and it orchestrates the exception handling actions of a set of agents in an application. This enables a uniform pattern for handling exceptional situations. *Global exceptions* are those programming exceptions handled by the guardian. When an exception of this kind is signaled by an agent, the guardian applies a corresponding *recovery 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 some local exception handling procedures in the agents impacted by the global exception.

Figure 3.1 illustrates a multi-agent application environment in the guardian model. Agents can notify and send exceptions to their guardian. In turn, the guardian,

according to the defined recovery rules and to the agents' contexts, can issue various kinds of commands to its agents or it may query the internal status of an agent.

One main drawback of the proposed approach, besides a strong centralization, is that it violates the autonomy requirement of the agent paradigm. Indeed, the guardian can both access the agents' internal state and direct their behavior by sending them specific commands.

3.3 Sentinels

A similar, but more decentralized approach w.r.t. to the guardian, was previously introduced in (Hägg, 1997). Here, *sentinels* are special agents introduced to realize a fault-tolerance layer. They are specialized in error detection and recovery, guarding certain functionalities and protecting the system from undesired states. They form a sort of control structure to the multi-agent system.

In the proposed approach, sentinels do not partake in domain problem solving, but they can intervene if necessary, choosing alternative problem solving methods for agents, excluding faulty agents, altering parameters for agents, and reporting to human operators.

It's worth noting that sentinels have the ability to inspect (parts of) the agents' internal states. *Checkpoints* allow to identify some of the agents' beliefs deemed accessible to a given sentinel. The author justify such an assumption, despite decreasing the agents' freedom, as necessary for preserving the system's integrity. When an exception is detected, the sentinels execute specific code to recover a desired state.

Even if such a decision is motivated by the fact that, depending on the application priorities, integrity could be put before autonomy, also the sentinel approach violates the agent paradigm, since sentinels can access and execute agent's internal code.

3.3.1 Sentinel-like Agents

The original sentinel-based approach presented by Hägg, 1997 has been extended in (Klein and Dellarocas, 1999; Dellarocas and Klein, 2000). The authors propose an approach based on a shared exception handling service providing sentinels with handling recipes inspired from the management research, to be plugged into existing agent systems. The approach comes from research in the context of workflow enactments (Klein and Dellarocas, 2000).

The proposed service actively looks system-wide for exceptions and prescribes specific interventions instantiated for the particular context from a body of general treatment procedures. The aim of the approach is then to instantiate generic exception handling expertise into specific situations.

The exception handling service communicates with agents using a predefined language for detecting exceptions (the query language) and for describing exception resolution actions (the action language). Agents can take any form as long as they are capable of responding appropriately to at least a minimum subset of the query and action languages. They are required, for their part, to implement a normative behavior plus a minimal set of interfaces to report on their own behavior and modify their actions according to the prescriptions.

When a new agent is introduced into the system, its normative behavior is mapped to a list of the *failure modes* that are known to occur for that kind of normative behavior, and sentinels are generated to detect those modes. Failure mode identification is done making use of a taxonomy of generic problem solving processes wherein each process is annotated with the different ways it can fail. Every failure is associated with a script that searches for the pattern of agent behavior corresponding to the failure. These scripts, once instantiated, play the role of sentinels, alerting the exception handling service when the condition they were created to detect has occurred.

Detected failures are then treated as “symptoms” by the exception handling service to diagnose specific exceptions based on a heuristic classification. The diagnosis

hierarchy is structured as a decision tree wherein the system starts at the top most abstract diagnosis and attempts to refine it to more specific diagnoses by traversing down the tree and selecting the appropriate decision branches by asking questions, expressed as query language statements, to the relevant agents.

Once one or more candidate diagnoses for an exception have been identified the system generates specific plans for resolving the problem, by using a knowledge base of generic exception resolution strategies. Strategies are represented as executable script templates whose actions are described using the action language, to be assigned to agents.

In (Klein et al., 2003) the authors propose to complete the system with a *reliability database*. Failing agents are registered in the reliability database to keep track of problems occurring with high frequency (e.g., an agent death). The database serves the purpose of guiding sentinels in recovery procedures to improve the mean recovery time. Moreover, in this updated proposal, each agent is associated with a sentinel which filters all of its in- and out-going message traffic.

3.3.2 Sentinels in Agent-based Grid Computing

Another extension to the sentinel-based model has been developed by Shah et al. to focus on an exception diagnosis mechanism for detecting when sentinel agents should react (Shah et al., 2004; Shah et al., 2005; Shah et al., 2006). In the proposed model, sentinel agents are provided by the MAS infrastructure and require the problem-solving agents to cooperatively provide information regarding their mental attitudes, whenever requested during the exception detection process. This enables the sentinel agents to diagnose exceptions interactively and heuristically by asking questions from affected agents through ACL messages. Exception diagnosis is realized by means of a heuristic classifier.

When an agent joins the MAS, a sentinel agent is created and assigned to it; all communication with the agent takes place via its associated sentinel. When an agent plans to interact with others, it sends ACL messages via its associated sentinel, which

detects any abnormalities in the messages and exploits them to diagnose the possible causes of occurring exceptions.

3.4 SaGE in MaDKit

SaGE (Souchon et al., 2003; Souchon, 2005) is an exception handling system for multi-agent programming that extends the MaDKit platform[4]. It integrates exception handling in the execution model of the agents. In MaDKit, agents hold some roles and provide services to each other according to the roles. Handlers can be defined and associated with the services provided by an agent, as part of its behavior. In particular, handlers associated with a service are designed to treat exceptions that are raised while executing the service. The objective of the service, its current state and the impact of exceptions on its completion can be taken into account when coding the handler.

Handlers can be associated directly with agents and roles, as well. Handlers associated with an agent are a practical means to define a single handler for all the services provided by the given agent. Handlers associated with a role are designed to treat exceptions that concern all agents which play a given role.

Exceptions are signaled, during the execution of services thanks to calls to a specific primitive. When an exception is signaled, the execution of the defective service is suspended. First, a handler searched locally, in the list of handlers associated with the service. If such a handler is found, it is executed. If not, the search carries on among the handlers associated with the agent that executed the defective service. If no adequate handler is found, the exception is propagated to the client agent that requested the service and the search continues iteratively.

Moreover, SaGE provides support for *concerted* exception handling (Issarny, 2001). Indeed, as advocated in (Campbell and Randell, 1986), the occurrence of many exceptions may be indicative of an exceptional state dependent upon the composition

[4] `http://www.madkit.net/`

of all the signaled exceptions. In SaGE, exceptions concurrently signaled by the entities participating to a collective activity can be composed together, by a *resolution function*, as a unique exception that is called concerted exception. This concerted exception is used instead of the individual exceptions raised by the participants to trigger handlers at the collective activity level.

To enable concerted exception support, propagated exceptions are not directly handled, but stored in a log which is associated to the recipient service. This log maintains the history of the so far propagated exceptions (along with information such as the sources of the exceptions). Whenever a new propagated exception is logged, the resolution function associated with the recipient service is executed to evaluate the situation. Depending on the nature and the number of logged exceptions, the function determines if an exception is to be effectively propagated. If so, the propagated exception can be the last propagated one or a new exception that is calculated from a set of logged exceptions, the conjunction of which creates a critical situation.

One main limitation of SaGE in MaDKit is that agents are assumed to be benevolent and cooperative, so the framework is not well suited for open systems where agents are self-interested.

3.5 An Agent Execution Model Encompassing Exceptions

The work of Platon et al., instead, is focused on proposing an architecture built upon a definition of exception explicitly encompassing the peculiarities of the agent paradigm: openness, heterogeneity and autonomy (Platon, 2007; Platon et al., 2007a; Platon et al., 2007b; Platon et al., 2008).

The authors propose the following definition of an agent exception, differentiating it from the traditional notion of programming exception:

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

Differently from programming-level exceptions, the source of an agent exception is not an event *per se*, which could be related to the call-back from an operation invocation that signals an exception. The source is instead the agent itself. It follows that exceptions make sense inside an agent; the exceptional character of an event depends on the point of view of each receiving agent. The decision criteria for exception is the expectation. An agent would be able to interpret an input as exceptional, whenever such an input does not match its expectations.

Moreover, Platon et al. point out that in a multi-agent system "there is no call stack to rewind with an exceptional event, which must be handled in the continuity of the agent activity." That is to say, a mechanism handling the failure cannot disrupt the execution cycle of the agent, as it happens for methods and actors raising exceptions.

An execution model that encompasses exception handling is then proposed so as to allow the agents to preserve control over themselves all along the execution and despite the occurrence of exceptions. In particular, the proposed architecture extends the agent perception and actuation components. The former is extended by adding an evaluation module consisting of some relevance and expectation criteria to classify input events (the 'percepts') and let the agent initiate potential exception management when required internally.

Events classified by the evaluation are forwarded to the agent internal mechanisms component. Here, an exception layer introduces appropriate mechanisms to deal with exceptions, so that the agent can continue its activity despite the occurrence of an unexpected event. Whenever exceptional conditions are detected, a handler selection process is initiated. The component as a whole manipulates the internal representation and its output is an action passed to the actuation for producing an effect in the environment.

Under this perspective, exception propagation is intended as the mechanism that describes how agents deal with exceptional situations they are unable to manage. 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 or not. By consequence, the use of this expression differs significantly from programming exceptions, where propagating an exception means 'passing' it along the call stack of the process until a handler is found.

Exception handling, in turn, is the actual processing of an exceptional situation by the 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).

3.6 Exceptions and Commitment-based Protocols

Taking a different perspective, in (Mallya and Singh, 2005b; Mallya, 2005) exceptions are modeled in the context of business processes via commitments-based interaction protocols. A *social commitment* $\mathsf{C}(x, y, p, q)$ denotes the fact that an agent x (debtor) commits towards an agent y (creditor) to bring about a consequent condition q if an antecedent condition p holds. Commitments have a well-defined lifecycle and are manipulated by the agents through a set of standard operations (Singh, 1999). A commitment protocol is defined in terms of the social commitments that can be created by the agents; their evolution determines the possible runs of execution.

The authors propose to deal with *expected* exceptions (i.e., deviations from the normal flow that occur often enough to be part of the model) by specifying a hierarchy of preferred runs. Preferences can be then used to define exceptional runs and agents can reason on such preferences to decide in which enactments to take part.

The paper proposes also an approach to deal with *unexpected* exceptions (i.e., exceptions that are detected at runtime, but are not part of the protocol model). Exception handlers, in this case, are treated as runs just like protocols. When an

exceptional run is detected, a handler is searched from a library of predefined ones. If a suitable one is available, it can be then spliced inside the given protocol (Mallya and Singh, 2005a).

The approach proposed by the authors seems promising, although some concerns related to scalability can be identified. Indeed, as the authors state, splicing exception handlers at runtime requires a search through a library of handlers, that can be computationally demanding. Conversely, inducing a preference structure over runs requires considerable design-time effort and extensive domain specific knowledge (which could be not always available).

3.7 An Obligation-based Approach

Still in the context of interaction protocols and of normative multi-agent systems, Gutierrez-Garcia et al., (2009) propose an approach for exception handling in interaction protocols, where both interaction protocols and exception handlers are modeled through *obligations*. Exceptions here are seen as abnormal situations in which agents cannot release an obligation. The obligation is canceled and, similarly to (Mallya and Singh, 2005b), a handler, to be spliced in the protocol, is sought for in a repository. Exception handlers are modeled as further obligations to be issued towards the agents.

The approach we will present in the following chapter leverages obligations, as well. Nonetheless, this approach differs from ours because: (1) it is not framed in an organizational dimension; (2) exceptions are not first-class objects, constituting a feedback for a failure, but are rather simply conceived as abnormal situations emerging during the enactment of an interaction protocol. Furthermore, the proposal is mainly theoretical, and no integration in any MAS platform is discussed.

3.8 Failure Handling in SARL

SARL (Rodriguez et al., 2014) is a general-purpose agent-oriented programming language and platform, which supports the notion of holonic multi-agent system (Gerber et al., 1999; Schillo and Fischer, 2002; Fischer et al., 2003). A multi-agent system in SARL is a collection of agents interacting together in a collection of shared distributed *spaces*, grouped into *contexts*. Each agent exhibits a collection of *capacities*, which may be then concretely implemented by various *skills*. An agent may be equipped with one or more *behaviors*, which map a collection of perceptions represented by *events* to a sequence of *actions*, coherent with the agent's capacities. An event is the specification of some occurrence in a space that may potentially trigger effects by an agent behavior.

At the language level, SARL supports exception throwing and catching within an agent's code, in a similar way to what done in Java (upon which SARL is built). Recently, the authors introduced a specific kind of event that represents any failure or validation error that an agent could handle, if interested. Each time an agent needs to be notified about a failure (e.g., during the execution of its tasks), an occurrence of this event type is fired in the internal context of the agent, which may then handle it through some suitable behavior.

At the same time, following the holonic perspective, agents can be composed of other agents. This allows one to define hierarchical structures, called *holarchies*. The SARL API provides dedicated functionalities for propagating (failure) events from one agent to its parent in a holarchy.

One major limitation of the approach is that no explicit responsibility assumption is made by the agents on the handling of possibly occurring failure events. In other words, no specific relationships are established among the agents w.r.t. the raising and handling of exceptions. For this reason, no rightful expectation can be created within the society of agents concerning if and how failures (i.e., exceptions) will be actually handled. At the same time, no guarantee is given about the fact that an

agent receiving a failure event will be equipped with the capacities to effectively tackle the situation.

3.9 Fault Tolerance in Jason

Jason (Bordini et al., 2007) is a well-known platform, implemented in Java, for the development of multi-agent systems based on the BDI-inspired language AgentSpeak(L) (Rao, 1996). A Jason agent is an entity composed of a set of *beliefs*, i.e., predicates representing the agent's current state and knowledge about the environment, a set of *goals*, which correspond to tasks the agent has to perform, and a set of *plans* which are courses of actions, either internal or external, triggered by events.

Goals are achieved by the execution of plans. It is possible to specify achievement (operator '!') as well as test (operator '?') goals. A plan, in turn, has the following structure:

$$triggering\_event : \langle context \rangle \leftarrow \langle body \rangle$$

triggering_event denotes the event that the plan handles (a belief/goal addition or deletion), while *context* specifies the circumstances in which the plan could be used. Together, the triggering event and the context are called the *head* of the plan, while the *body* expresses the course of action that should be taken.

The interpretation of the agent program realizes the agent's *reasoning cycle*. An agent constantly perceives the environment, reacting to events and reasoning about how to act so as to achieve its goals, then acting so as to change the environment. This cyclic behavior is done according to the plans available to the agent in its *plan library*.

In the following, we briefly review two approaches addressing fault-tolerance in Jason. The former, based on the notion of contingency plan, is built-in in the Jason platform and it is conceived to address failures which occur locally to a given agent. The latter approach, in turn proposes an extension of Jason in order to allow the

specification of monitoring and supervision relations among agents to address faults involving multiple parties.

3.9.1 Contingency Plans

Despite not explicitly encompassing the notion of agent exception, Jason provides a mechanism to deal with failures which are local to a given agent, i.e., failures occurring during the execution of plans. Such a mechanism follows the line drawn by the execution model proposed by Platon et al., presented in Section 3.5. A programmer can define dedicated *contingency plans* (i.e., handlers) specifying the course of actions to undertake when a plan fails to achieve the goal it was supposed to achieve.

Following (Bordini et al., 2007), three main causes for plan failures can be identified:

Lack of relevant or applicable plans for an achievement goal This is the case in which a plan being executed requires a sub-goal to be achieved, and the agent cannot achieve that.

Failure of a test goal This represents a situation in which the agent 'expected' to believe that a certain property of the world holds, but in fact the property does not hold when required.

Action failure If an action fails, the plan where it appears also fails.

While in an agent's code standard triggering events have the form `+!g` for plans to achieve goal `g`, `-!g` denotes the triggering event for the contingency plan to apply when a plan for `+!g` fails. Regardless of the reason for a plan failure, the Jason interpreter generates a goal deletion event `-!g` if the plan for a corresponding goal achievement `+!g` has failed. If available, the contingency plan is then triggered to 'clean up'.

3.9.2 Monitoring and Supervision in eJason

As a final note, we briefly review the proposal presented in (Fernández-Díaz et al., 2015), where eJason, an Erlang-based extension of Jason with constructs for fault tolerance, is proposed. Two fault tolerance mechanisms are introduced, allowing detection and recovery of faults occurring in the interaction between multiple Jason agents.

The former, called *monitoring*, can be used when some agent a is interested in the execution state of another agent b. If a starts a monitoring relation with b, a will know whether b dies, whether it leaves the system, whether it is restarted, whether it does not belong to the system at all, or whether it is added to the system (in case it was not yet). A monitoring relation does not impose any kind of additional responsibility to neither the monitoring nor the monitored agents.

The *supervision* mechanism, in turn, allows the implementation of possibly complex fault recovery behaviors. A supervisor agent can start a supervision relation with some set of agents. Then, the supervisor acquires the responsibility of carrying out certain fault recovery actions, according to a supervision policy, in order to maintain (up to some extent) the availability of the supervised agents. The supervisor agent handles requests from other agents that suspect the failure of some of its supervised agents. Finally, a supervisor agent receives the ability to execute some fault detection (ping) and fault recovery (restart, unblock, revive) actions over its supervised agents. The supervision policies define the courses of actions of the supervisors with regards to both fault identification and recovery. An example of a valid supervision policy is one that requires pinging the supervised agents every 15 seconds and restarting any of them that fail to answer to a ping two consecutive times.

The proposed mechanism bears strong similarities with the fault tolerance model adopted in Akka (see Section 2.3). However, while in the actor model the establishment of supervision relations can be justified by the fact that a hierarchy exists among parents and children actors, in the proposed model some concerns emerge on how supervision relations among agents are created. Indeed, the platform allows

a supervisor agent to establish a supervision relation with another agent without any agreement from the supervised side. As a consequence, the supervisor is granted powers to execute actions that affect the supervised agent's lifecycle (e.g., restart), resulting in a clear violation of the autonomy principle.

4 A Proposal for Exception Handling in Multi-Agent Systems

Contents

In this chapter, we present an abstract architecture for handling exceptions in an multi-agent setting, grounded on the notions of *responsibility* and *feedback*. The proposal relies on the same high-level abstractions that characterize the multi-agent paradigm, thereby not interfering with the agents' autonomy.

4.1 Challenges and Open Issues

As highlighted in the previous chapters, the need for exceptions emerges from the desire of structuring and modularizing software, separating concerns into components that interact. Exception handling enables robust software composition because perturbations occurring in one component may affect the ones interacting with it, as well. It systematizes the way in which feedback about a perturbation detected in one component is delivered to, and must be addressed by, the others. This has a positive impact in terms of generality, because the most appropriate recovery to be applied may vary, depending on the exception receiver's purposes. Under this perspective, multi-agent systems bring software structuring, modularization, and separation of concerns to an extreme – agents are autonomous and less strictly coupled than operations in sequential programs or even actors in the actor model. Still, autonomous agents cooperate and rely on one another to pursue their aims. Surprisingly, however, no consensus has been reached in the field with respect to a model for exception handling.

The proposals in the literature, reviewed in the previous chapter, try to fill in this gap, but many of these approaches address exception handling in a way that interferes with one of the basic principles of the agent paradigm: the autonomy of the agents. Furthermore, importantly, no proposal conceives exception handling as originally postulated in the seminal work of Goodenough. As a result, current conceptual models for multi-agent systems fall short in delivering a fully-fledged exception handling mechanism.

As pointed out in Section 2.2, Goodenough's work on programming languages brings forward two fundamental aspects of exception handling:

1. It always involves two parties: a party that is *responsible* for raising an exception, and another party that is *responsible* for handling it;
2. It captures the need for some *feedback* from the former to the latter that allows coping with the exception.

Notably, exception handling mechanisms adopted in programming languages and in the actor model reflect this vision. Paralleling Goodenough's approach, it is easy to see a parent actor as an operation invoker and a child actor as the invoked operation. Children actors are the ones responsible for raising exceptions when perturbations are detected, while the parent actor is the one responsible for handling them, since it is the one which can determine the impact of a specific failure of one child onto the concurrent execution of the others. The exception object passed from the child to the parent represents the feedback allowing the parent to choose which strategy to undertake to handle the failure.

A substantial difference between MAS and actors (or operations in a program) is that agents are not structurally bound by parent-child (or invoker-invoked) relationships. Thus, when an agent meets a failure, it cannot easily determine which other agent could handle the related exception. The agent that failed may ask the other agents but, due to autonomy, it would not be guaranteed that its request would ever be considered. The other agents, that are endowed with the right abilities, may prefer to achieve some other goal, and, in general, the system will not provide the means for persuading them to act otherwise. More importantly, the agents capable to solve the problem may lack the relevant information needed for handling the exception at hand in an effective way.

Broadly speaking, what multi-agent systems currently lack is a *clear distribution of responsibilities* among agents for raising and handling exceptions. As a result, current models do not typically encompass conceptual as well as programming tools to allow the relevant *feedback* concerning the exceptional situation flow, through appropriately devised channels, from the component (i.e., agent) involved in it to the right components equipped with the means (capabilities, resources, willingness, etc.) for handling it.

4.2 Exception Handling as Responsibility

Despite the challenges outlined above, we believe that Multi-Agent Organizations (MAOs) could provide the structure we need. Since multi-agent organizations are built upon responsibilities, we claim that exception handling – in essence, a matter of responsibility distribution – can be integrated seamlessly.

Key features of many organizational models are a functional decomposition of an organizational goal and a normative system. Each agent taking part in an organization is required to take on responsibilities for some parts of the organizational goal, whose achievement is distributed among the agents. The normative system enables the orchestration of the activities generating *obligations* towards agents to execute tasks according to the functional decomposition and to the responsibilities taken by them.

However, agents may fail the expectations put on them. From this perspective, we can interpret an exception in a multi-agent organization as:

> *An event which denotes the impossibility, for some agent, to fulfill one of its responsibilities – e.g. the failure in the execution of a task or a missed deadline.*

We then propose to leverage on responsibility not only to model the duties of the agents in relation to the organizational goal, but also to enable agents to provide feedback about exceptions, occurring within the organization operation, and to identify those agents entitled for handling them.

When agents join an organization, they will be asked to take on the responsibilities not only for the organizational goals, but also:

1. For providing feedback about the context where exceptions are detected while pursuing organizational goals;
2. If appointed, for handling such exceptions once the needed information is available.

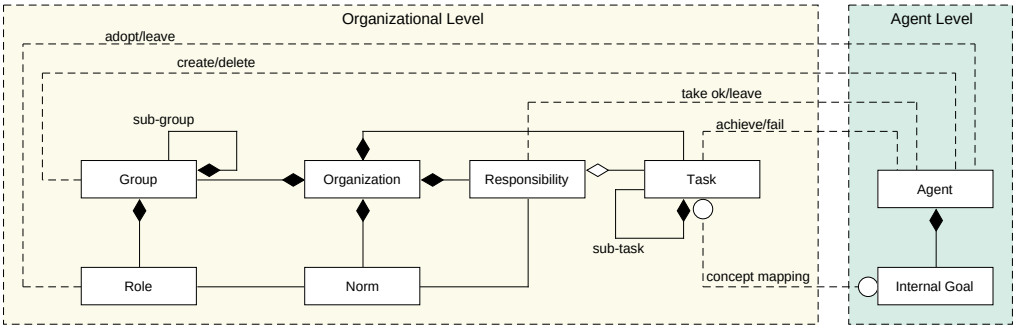

Figure 4.1: Abstract model of an agent organization.

Responsibilities, thus, become a tool to define the scope of the exceptions, expressed with respect to the organizational state, that agents ought to raise or handle during the achievement of the organizational goal(s).

Taking as references the conceptual models of organizations and institutions discussed in (Boissier et al., 2013; Feltus, 2014; Brito et al., 2017; Zambonelli et al., 2003; Dignum et al., 2004a; Dignum et al., 2004b; Hübner et al., 2007), we have distilled a general exception handling mechanism for organizational settings.

We now briefly explain the concepts commonly underpinning organizational models, summarized in Figure 4.1 and then present the main concepts at the basis of our proposed exception handling mechanism, together with their usage.

4.3 Multi-Agent Organizations

To face the inherent need of coordination among autonomous agents, the *organization* metaphor has been used for a long time in MAS research. The organization is, in fact, a useful mechanism for modularizing code spread over different software components that are opaque and independent of each other.

Multi-agent organizations (MAOs) represent strategies for decomposing complex organizational goals into simpler sub-tasks and allocating them to roles. By adopting roles in the organization, agents acquire responsibilities and execute the corresponding tasks in a distributed, coordinated and regulated fashion.

Agent organizations show the same kind of structure and of advantages that sociologist Dave Elder-Vass explains for human organizations: an organization provides a structure of constraints that allows a system consisting of many parts to act as a whole, with the aim of achieving goals that otherwise would not be achievable (or not as easily) (Elder-Vass, 2011).

4.3.1 Tasks, Responsibilities and Roles in MAOs

Many methodologies for multi-agent organizations (e.g., (Zambonelli et al., 2003; Dignum et al., 2004a; Hübner et al., 2007)) hinge on the concepts of `Task` and of responsibility about some task. When looking back, a set of initial proposals (Corkill and Lesser, 1983; Dignum, 2009) have defined an explicit structure of `Roles` and relations, through which responsibilities of tasks are distributed by adoption of roles, among the agents participating in the organization.

In this sense, the notion of `Responsibility` can assume several nuances of meaning. In (Vincent, 2011), an ontology relating six different responsibility concepts (capacity, causal, role, outcome, virtue, and liability) is proposed. Roughly speaking, they respectively amount to: doing the right thing, having duties, an outcome being ascribable to someone, a condition that produced something, the capacity to understand and decide what to do, something being legally attributable.

From a computational point of view, the notion of responsibility has been used in a wide number of methodologies for the design of multi-agent systems. For instance, in Gaia (Zambonelli et al., 2003), responsibilities are used to model what agents ought to do within the system; the functionality of a role is defined by its responsibilities. The OperA framework (Dignum et al., 2004a) is able to define the global aims of an organization (tasks) and the objectives and responsibilities of its participants. In this acceptation, however, a responsibility is a duty that falls on an agent. When an agent joins a system, thus, it could be asked to perform tasks without prior knowledge about them. We take a different perspective: agents take on their responsibilities by means of an explicit declaration. As said, an organization is a structure to coordinate

the distributed execution of tasks for reaching complex objectives. In our proposal, when agents join an organization, they accept explicitly the responsibility of some of its tasks.

However, these models are well adapted to "closed agent organizations" where benevolent agents, always complying, coordinate with each other to achieve their responsibility assumptions (actions, goals or interactions).

4.3.2 Normative Organizations

A second generation of models, following the electronic institution pioneering approaches (Esteva et al., 2001), has introduced `Norms` in the structure of roles, tasks and responsibilities, giving birth to *normative multiagent systems* (Boella et al., 2006; Boella et al., 2008). These social coordination frameworks (Aldewereld et al., 2016) have the potential to target open systems. Thanks to a set of norms, the structures of distributed responsibilities among agents have been enriched with structures of *social expectations*: besides being the source of task responsibility assumption, roles have become the anchoring point of social expectations on the behavior of the agents who will play them.

In other words, norms allow to shape the expected behavior of the system w.r.t. the responsibilities of the participating agents towards the distributed tasks. Through, e.g., commitments, authorizations, prohibitions (Singh, 2013), norms can yield obligations about the tasks that agents are held to fulfill; they are, therefore, used to describe the ideal behavior of agents in terms of their responsibilities, rights and duties (López y López and Luck, 2003).

As for normative MAS, *normative organizations* (Dignum, 2004; Dignum et al., 2004b; Fornara et al., 2008; Dastani et al., 2009; Boissier et al., 2013) are typically equipped with a set of mechanisms to publish, enact, adapt, monitor, and enforce normative behaviors. Thus, once decided to adopt a role with the accompanying norms and thus participating to the organization, agents assume the responsibility of the targeted tasks. This assumption of responsibility corresponds to an agent's

declaration of being recipient for (and hence moved by) some organizational events. In particular, these events amount to the *obligations* that the normative system of the organization issues when agents are expected to perform their tasks.

In OMNI (Dignum et al., 2004b) and $\mathcal{M}$OISE (Hübner et al., 2007), for instance, a functional decomposition describes how a complex goal (task) can be achieved in a distributed way. Agents joining the organization are expected to contribute by achieving sub-goals of such a decomposition, whenever obligations are triggered by the organization towards them. In $\mathcal{M}$OISE, agents are held to explicitly commit to missions (i.e., subsets of goals), this act implies a taking of responsibility of the agents towards their missions and, hence, the acceptance of the related obligations that will be issued by the organization.

In short, all these frameworks see responsibilities as duties related to certain organizational tasks that an agent has to accomplish while taking part in an organization, and use obligations as a mechanism for coordinating agents in discharging their responsibilities, i.e., accomplishing their tasks. The agent autonomy is preserved, since agents can reason about the normative system, and hence can deliberate how to act once obligations are triggered; they can even decide not to satisfy an obligation. Thus, norms originate responsibilities in organizations (Feltus, 2014), as well as in institutions (López y López and Luck, 2003), and responsibilities aggregate the tasks that ought to be performed within the society of agents. As a result, a role becomes a way to model, through norms, the powers and duties of its players within the context of the organization. An `Agent` can adopt and leave a role, take and leave a responsibility, and can achieve or fail a task by internalizing it as a goal of its own ones.

Moreover, agents are rightfully expected by the organization to accomplish their duties. In case of violation, they may be enforced to do so through `Sanctions`. Sanctions are intended as deterrents to prevent norm violation, that is, to keep the execution oriented towards the achievement of the organizational goal.

The problem is that when the system faces an exceptional situation and some agent *fails* to complete a task, sanctions are of little utility, if any (Chopra and Singh, 2016;

Baldoni et al., 2018b). In this case, in fact, the agent may have earnestly tried its best to do what expected, but something which is not under its control hindered the achievement. To have an intuition, an agent may fail to deliver a parcel because a tree that fell blocks the only way that allows to reach the recipient. Moreover, sanctions are not generally accompanied by any feedback and feedback handling mechanisms oriented towards recovery, and thus they do not provide the right means to support robustness.

4.4 Introducing Exceptions

The main idea behind our proposal consists of reviewing the basic mechanism of exception handling in terms of responsibilities. To understand how the notion responsibility can be used for this purpose, we need to observe that an integral part of any exception handling mechanism is the bridge created between exception detection and exception handling independently of the components where the two events occur. We propose to realize such a bridge through properly devised responsibilities. When agents join an organization, they will be asked to take on the responsibilities not only for the organizational tasks, but also for rising exceptions when they encounter problems in fulfilling such responsibilities, and for handling some of the exceptions raised from others. As responsibilities "to do something", also responsibilities "to raise and handle exceptions" originate from the rules that govern the organization (i.e., the norms), and as such are associated with obligations that the organization issues whenever necessary.

In the rest of this chapter, we explain how these new sets of responsibilities can be integrated within the organizational model described above, and how they enable the realization of an exception handling mechanism that operates at the same level of abstractions of the agent paradigm.

Figure 4.2 reports the abstract model of a multi-agent organization extended with the main concepts at the basis of our exception handling mechanism.

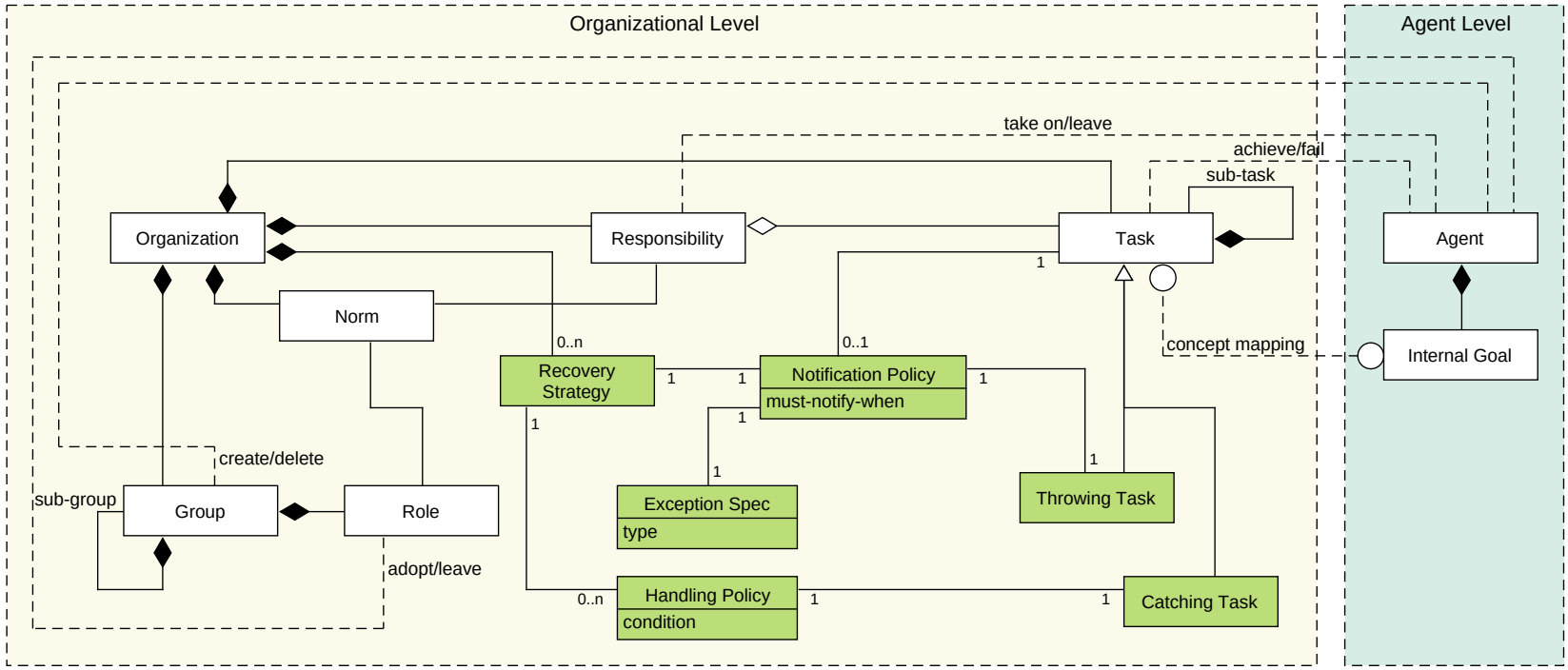

Figure 4.2: Abstract model of an agent organization extended for exception handling.

4.4.1 Recovery Strategies

In the proposed model, exception handling is built upon the concept of `Recovery Strategy`. Recovery strategies concretely realize the bridge between the agent(s) responsible for raising a given exception and the one(s) responsible for handling it.

For each exception deemed to possibly occur in the organizational functioning, we then define a related recovery strategy targeting it. In other words, a recovery strategy encodes when and how a given exception is to be raised and handled within the organization and shapes the resulting responsibilities accordingly.

It is associated with a notification policy and one or more handling policies, governing, respectively, the raising of an exception and one (or more) task(s) to properly tackle it.

4.4.2 Notification Policies and Throwing Tasks

Each recovery strategy encompasses a `Notification Policy` for the related exception. The notification policy specifies the circumstances in which the agent responsible for throwing the exception will be asked to do so. In particular, each notification policy is characterized by a `must-notify-when` attribute and it is associated with a `Throwing Task`, an `Exception Spec`, and a `Task`.

The association between notification policy and task captures the object of the exception to be raised. That is, the exception is related to a perturbation occurring in an agent's fulfillment of its responsibility concerning the given task.

The `must-notify-when` attribute denotes the kind of situation amounting to the perturbation to address through exception handling, i.e. the condition which triggers the notification policy. It can be conceived as a condition expressing the state of the organization in which an exceptional situation occurs, thereby causing the activation of the notification policy. In our model, such an occurrence of an exceptional situation requires the throwing of an exception by some agent, in order to obtain

the information (feedback) needed for recovery. To this end, each notification policy includes a `Throwing Task`.

A throwing task denotes the task at an organizational level of raising an exception. Being a task, it is subject to responsibility assumption by the agents. It's purpose is to push the agent, by discharging its responsibility, provide relevant feedback about the perturbation, so as to enable a successful recovery.

Notification policies bring along normative expectations that can be formulated according to the normative layer. Indeed, norms delimit the scope of the responsibilities concerning throwing tasks, too. The `must-notify-when` condition, in particular, specifies when an obligation to pursue the throwing task is to be issued towards the responsible agent(s).

It's worth noting that throwing tasks might be structured ones, involving multiple sub-tasks and the collaboration of multiple agents to collect and make available all the relevant information concerning the exceptional situation.

4.4.3 Exception Spec

The notion of `Exception Spec` captures the shape of the piece of knowledge, i.e., the feedback, that the agent(s) responsible for throwing the exception has to provide to the agent(s) in charge for handling it. Indeed, the throwing of an exception is the result of the mapping of the corresponding throwing task to an agent's internal goal. The result of pursuing such an internal goal is a set of facts that gain a social meaning as an exception, and must follow the structure specified by the exception spec.

To give an intuition, we can conceive an exception spec as a form to be filled by the agent in charge of executing the corresponding throwing task. The filled form amounts to the actual exception that is thrown and is made available to the handler agent.

It's worth noting that throwing an exception that follows a given exception spec allows the thrower agent to disclose some local information, deemed relevant for exception handling. An explicit responsibility assumption regarding a throwing task thereby creates a social expectation regarding the fact that the agent will be actually able to provide such information, when requested to do so.

4.4.4 Handling Policies and Catching Tasks

`Handling Policies` realize the second part of the above mentioned bridge, shaping the courses of action to follow in order to handle an exception, should it be thrown during the organization functioning.

Each recovery strategy encompasses one or more handling policies. Each handling policy, in turn, is characterized by a `condition`, denoting the states of the world in which the policy is applicable, and it is associated with a `Catching Task`. In analogy with throwing tasks, catching tasks denote the task, at an organizational level, of handling a previously thrown exception.

More precisely, a given catching task is to be assigned for achievement to its responsible agent once the exception targeted by the enclosing recovery strategy has been thrown and when the condition specified by the handling policy is verified.

A recovery strategy might include several handling policies. The rationale behind this choice is that the occurrence of an exception might have to be handled in different ways in different circumstances. To give an insight, a fire should be put out either with or without water depending on the presence of electrical components. Multiple handling policies with different activation conditions serve this purpose.

Similarly to throwing tasks, catching tasks are targets of responsibility assumption by the agents. Since a catching task models the course of action to handle an exception, it may amount to a complex one and may involve multiple agents, as well.

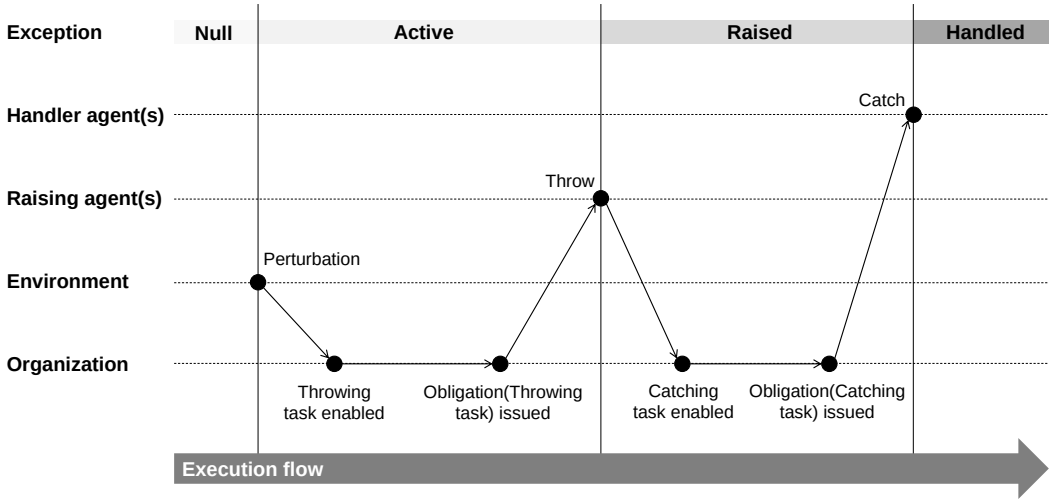

Figure 4.3: Lifecycle of an exception in our proposed model.

4.5 Exception Handling in Operation

The diagram in Figure 4.3 illustrates the runtime lifecycle of an exception, in our proposed model. As soon as a perturbation targeted by a given recovery strategy occurs in the environment, the corresponding exception becomes *active*, and the exception handling process is triggered. According to the specified notification policy, the occurence of the perturbation enables, at the organizational level, one (or more) throwing task(s) and the related obligation(s) is issued towards the responsible agent(s). The throwing action performed by such raising agent(s) concretely instantiates the exception, which then becomes *raised*. At this point, in accordance with the handling policies included in the recovery strategy, obligations are issued concerning the enabled catching tasks. The exception becomes finally *handled* as soon as it is caught by the handler agent(s), i.e., the one(s) responsible for the catching tasks.

More in detail, at runtime, exception handling is practically enforced within an organization, leveraging the model above, as follows:

1. Agents take part in the organization by adopting some roles and are required to take on the responsibilities induced by their roles explicitly. These respon-

sibilities may concern “standard” tasks, as well as throwing and catching tasks.

2. As soon as all the agents have taken on their responsibilities, the distributed execution is regulated through norms. Norms enable the coordination of the agents by determining when obligations concerning tasks must be issued towards the responsible agents.

3. If a perturbation occurs, denoting the impossibility for some agent to fulfill one of its responsibilities, a recovery strategy is searched to address it. The recovery strategy must include a notification policy applicable to the exceptional situation at hand, i.e., it’s `must-notify-when` condition must amount to the perturbation at hand. This denotes the presence of an active exception that must be coped with.

4. The throwing task included in the selected notification policy is enabled and an obligation concerning it is issued. To fulfill its responsibility, the responsible agent is then required to throw an exception compliant with the exception spec defined within the notification policy. The exception instance that is thrown constitutes the feedback concerning the perturbation, coming from an informed source.

5. As soon as the exception is raised (i.e., the feedback is available) an applicable handling policy is selected among the ones defined within the scope of the previously selected recovery strategy. If multiple handling policies are applicable to the exception at hand, all of them are applied. The corresponding catching tasks are enabled.

6. Obligation are issued concerning the enabled catching tasks. The responsible agents can leverage the information encoded by the exception thrown beforehand to put in place the most appropriate countermeasures to handle the exception, fulfill their responsibilities and recover.

It is worth noting that the mechanism allows to capture a wide range of situations, which are summarized below.

4.5.1 Exceptions Raised Collectively

The raising of an exception may involve multiple agents. In particular, we can have two cases (possibly combined together).

First, throwing tasks may be composite ones, involving multiple sub-tasks to be assigned to different agents. This allows to model the fact that the raising of an exception may be a process composed of multiple steps, requiring the collective collaboration of multiple agents. To produce the required feedback, for instance, it could be necessary to, first, gather relevant data, second, elaborate it, and finally deliver it in the right format. All these actions may fall under the responsibility of different agents.

Second, at the same time, since a given role may be played by multiple agents, should this happen, the responsibility for a given task would be taken by all the role players. This holds for throwing tasks, as well. Consider, for instance, a perturbation causing the failure of a task. All the agents playing a specific role may be required to provide a testimony, either because directly involved in the failure or because equipped with a useful expertise.

In other words, notification policies can be effectively used as a tool to model exceptions which have to be collectively raised by multiple agents, too. This could be due to the fact that the throwing task is complex and cannot be achieved in isolation or because the responsibility for it is shared among multiple agents.

4.5.2 Exceptions Handled Collectively

What explained above holds for catching tasks, as well, and enables the definition of handling policies to enforce the collective handling of a given exception by a set of agents. Again, multiple agents may be involved in exception handling because of the complexity of the catching task or because being players of the same role.

A catching task composed of several sub-tasks, carried out by different agents, allows to model structured courses of action to be put in place while handling an

exception. Moreover, the decomposition of a catching task allows one to involve into its execution different agents, which may be, in principle, equipped with different capabilities and know-how, and thereby able to contribute to the recovery in different ways.

Still as before, the same catching task may fall under the responsibility of many role player agents. This could happen when, in order to handle the exception effectively, the same (set of) action(s) is to be performed by many agents. Consider, for instance, in a company, an exception denoting the presence of a malware in the IT infrastructure. To handle the exception and solve the problem, all employees may be required to perform an antivirus analysis over their assigned laptop.

4.5.3 Recurrent Exception Handling

Perturbations may also occur during the raising and handling of exceptions. In other words, exceptions may be nested inside each other. In programming languages like Java, for example, this eventuality is captured by the fact that exceptions can be thrown also inside catch blocks.

Our exception handling mechanism allows to model this aspect, as well. Indeed, we can easily define recovery strategies targeting perturbations which affect the execution of further recovery strategies. For instance, a recovery strategy may be specified so as to address the impossibility for some agent to achieve a previously defined throwing or catching task.

Moreover, this approach enables the definitions of structured channels through which the feedback constituting an exception can flow from one agent to another and be used as a basis to produce further meaningful feedback (exceptions) to be propagated throughout the system.

In Chapter 5 we illustrate, as a use case, how the proposed exception handling architecture has been concretely realized in the context of the well-known JaCaMo framework for multi-agent organizations.

5 Case Study: the JaCaMo Framework

Contents

JaCaMo is a well-known framework for the development of multi-agent systems and organizations. For its good theoretical foundation and infrastructural maturity, it is one of the most widely used frameworks for multi-agent organizations programming. Nonetheless, its organizational model does not include any mechanism for handling exceptional situations at the organizational level – i.e., situations in which some agent does not achieve an organizational goal, thereby being unable to fulfill one of its responsibilities. This makes JaCaMo organizations fragile because the failure by an agent could in principle affect the achievement of the overall organizational goal(s). When a given goal is not achieved, no support is provided for recovering

from the exceptional situation except issuing additional obligations concerning the same goal towards the failing agents.

In this chapter we describe how the proposal presented in the previous chapter has been realized in the context of the JaCaMo framework. After a short introduction to JaCaMo, we show how we mapped our abstract model into JaCaMo's organizational metamodel and how we extended its infrastructure so as to encompass an exception handling mechanism based on responsibility. To practically illustrate the usage of the mechanism, we leverage the *ATM* example, already introduced in Chapter 2.

5.1 JaCaMo Basics

JaCaMo (Boissier et al., 2013) is a conceptual model and programming platform that integrates three different multi-agent dimensions: agents, environments and organizations. The agent dimension is used to program the individual, interacting, autonomous entities. The environment dimension is used to develop shared resources and connections to the real world. Finally, the organization dimension allows the structuring and regulation of complex interrelations and coordination between the agents and the shared environment.

5.1.1 Jason, CArtAgO and $\mathcal{M}$OISE

JaCaMo is built on top of three platforms: Jason (Bordini et al., 2007) for developing agents, CArtAgO (Ricci et al., 2009) for programming environments and $\mathcal{M}$OISE (Hübner et al., 2007) for programming organizations. As underlined by the authors, the aim of the framework is not only to technologically integrate the cited platforms, but also to integrate the related programming metamodels in order to simplify the development of complex multi-agent systems. This approach results in the realization of a high-level first-class support for developing agents, environments and organizations in synergy. It's interesting to point out that JaCaMo is one of the

first works aiming at investigating the integration of these three dimensions from both a design and a programming point of view.

Jason, already introduced in Section 3.9, is a platform for the development of multi-agent systems based on the BDI-inspired language AgentSpeak(L) (Rao, 1996). An agent is an entity composed of a set of beliefs, representing the agent's current state and knowledge about the environment, a set of goals, which correspond to tasks the agent has to perform, and a set of plans which are courses of actions, either internal or external, triggered by events, that can be taken by the agent in given circumstances. An agent's belief base is composed of a set of ground first-order atomic formulas. A Jason plan, in turn, is specified as:

$$triggering\_event : context \leftarrow body$$

where the *triggering_event* denotes the event that the plan handles (which can be either the addition or deletion of some belief or goal), the *context* specifies the circumstances when the plan is applicable, and the *body* is the course of action that should be taken.

CArtAgO is a framework and infrastructure for environment programming which conceives the environment as a first-class abstraction and a computational layer encapsulating functionalities and services that agents can explore and use at runtime (Weyns et al., 2007). It is based on the Agents & Artifacts metamodel (Omicini et al., 2008). Software environments are programmed as a dynamic set of *artifacts* (programmed in Java) collected into workspaces, possibly distributed among various nodes of a network. An agent can perceive the observable state of an artifact, reacting to events, and can act upon it by performing actions that correspond to operations provided by an artifact's usage interface.

Finally, $\mathcal{M}$OISE implements a programming model for the organizational dimension. It is based on the notions of roles, groups, schemes, goals, missions, and norms. It includes an organization modeling language for specifying multi-agent organiza-

tions and an organization management infrastructure for concretely managing the functioning of specific organization instances.

5.1.2 Organizational Specification

$\mathcal{M}$OISE's organizational model explicitly decomposes the specification of an agent organization into three further dimensions (Hübner et al., 2010b). The *structural* dimension specifies roles, groups and links between roles in the organization. The *functional* dimension encompasses one or more schemes that elicit how the global organizational goals are decomposed into sub-goals and how these sub-goals are grouped in coherent sets, called missions, to be distributed to the agents. The *normative* dimension binds the previous two by specifying the role permissions and obligations for missions.

Roles can be aggregated into groups and an instance of an organization can be composed of several groups. Similarly more than one scheme instance can be associated with an organization. In order to be executed, a given scheme instance must be assigned to one or more groups that become, then, responsible for it. Agents, in fact, are held to explicitly commit to the missions defined in the scheme, thereby taking responsibility for mission goals.

Organizational goals are mapped into individual agents' goals which agents can deliberate whether to achieve or not. This delegation is made by means of obligations according to the normative specification of the organization and the current state of the system. In this sense, an obligation is fulfilled when the corresponding goal is achieved by the recipient agent before a given deadline.

Listing 5.1 shows the functional specification of an organization realizing the *ATM* application of Example 1[5].

```
1 <functional-specification>
2         <scheme id="atm_sch">
3             <goal id="withdraw">
4                 <plan operator="sequence">
5                     <goal id="obtainAmount">
```

[5] In JaCaMo, organization specifications are written in XML.

```
                        <plan operator="sequence">
                            <goal id="getAmountAsString" />
                            <goal id="parseAmount" />
                        </plan>
                    </goal>
                    <goal id="provideMoney" />
                </plan>
            </goal>
            <mission id="mGetAmountAsString" min="1" max="1">
                <goal id="getAmountAsString" />
            </mission>
            <mission id="mParseAmount" min="1" max="1">
                <goal id="parseAmount" />
            </mission>
            <mission id="mObtainAmount" min="1" max="1">
                <goal id="obtainAmount" />
            </mission>
            <mission id="mProvideMoney" min="1" max="1">
                <goal id="provideMoney" />
            </mission>
            <mission id="mWithdraw" min="1" max="1">
                <goal id="withdraw" />
            </mission>
    </scheme>
</functional-specification>
```

Listing 5.1: Functional specification of a $\mathcal{M}$OISE organization realizing the *ATM* scenario.

Here, each component is modeled as an autonomous agent taking part in the organization. The overall goal of completing a withdrawal is reified into a single scheme, that decomposes it into several sub-goals to be achieved in sequence. First, the amount must be obtained from the user, and then the money provided. The amount of money is collected first as a string and then parsed by a dedicated parser agent. Such goals are then grouped into missions to be assigned to agents playing specific roles.

Figure 5.1, in turn, shows an excerpt of the JaCaMo programming metamodel by explicitly representing the dependencies and mappings between the main abstractions belonging to different dimensions. In particular agents' external actions are mapped into artifacts' operations and artifacts' observable properties and events into agents' beliefs. At the same time, the organizational infrastructure is designed as part of the environment in which agents are situated.

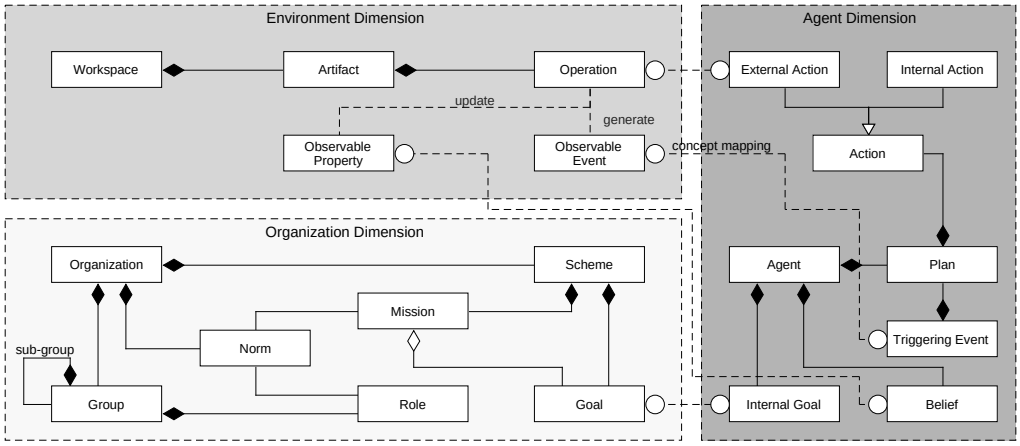

Figure 5.1: JaCaMo programming metamodel, as presented in (Boissier et al., 2013).

5.1.3 Organization Management Infrastructure

In JaCaMo, the different computational entities that manage the current state of an organization in terms of groups, schemes and normative states, which together constitute the organizational entity, are reified in the environment by means of some organizational artifacts, which encapsulate and enact the behavior described in the organizational specification. Moreover, these organizational artifacts provide the operations to be used by the agents to take part in an organization and act upon it, and the observable properties that make the state of the organization perceivable by the agents along with its evolution.

The main artifacts constituting the infrastructure are:

OrgBoard artifact, which keeps track of the current state of the organizational entity overall, one instance for each organization. It provides functionalities to create and delete groups and schemes according to a particular specification.

GroupBoard artifact, which manages the lifecycle of a specific group of agents, one for each group. For example, it provides the operations to adopt a role in a group, leave a given role, and add a scheme that the group will be responsible for.

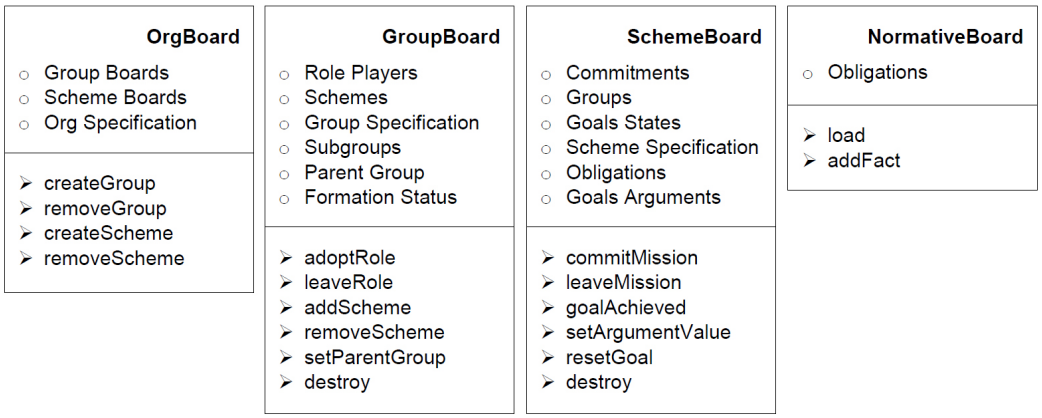

Figure 5.2: Basic kinds of organizational artifacts in JaCaMo and their usage interfaces.

SchemeBoard artifact, which manages the execution of a social scheme, one for each scheme, including the commitment to missions and the achievement of goals.

NormativeBoard artifact, used to maintain information concerning the agents compliance or not according to permissions and obligations defined between roles and missions.

Figure 5.2 shows an abstract representation of the organizational artifacts described above along with their usage interfaces (i.e., observable properties and operations available to agents).

During the execution of a scheme, the goals constituting it can be in three different states: *waiting*, *enabled*, or *satisfied*. The initial state is waiting, indicating that the goal cannot be pursued yet because it depends on the achievement of other goals (called preconditions) or the scheme is not yet well-formed, meaning that not all of the needed agents have committed to missions, yet. The set of precondition goals are deduced from the functional decomposition of the scheme. When the goal is ready to be pursued, it becomes enabled. As soon as a goal becomes enabled, the obligations to pursue it are issued towards the agents committed to a mission including the given goal. Finally, once achieved by the responsible agents, it becomes satisfied.

Listing 5.2 shows an excerpt of possible implementation of a *parser* agent, responsible for goal `parseAmount` in the *ATM* organization.

```
+obligation(Ag,_,done(_,parseAmount,Ag),_)
     : .my_name(Ag)
    <- !parseAmount;
       goalAchieved(parseAmount).

+!parseAmount
    <- parseAmount. //Operation over an environment artifact
```

Listing 5.2: Excerpt of the *parser* agent in the *ATM* scenario.

The agent is equipped with a plan triggered when the obligation to pursue goal `parseAmount` is issued towards it. By this, the obligation is mapped to an internal goal. The goal is practically pursued by the agent by executing some operations over the environment. In case of success, the organizational goal is set as as achieved, by means of the `goalAchieved(...)` primitive made available by the SchemeBoard artifact.

5.1.4 Normative Programming

At runtime, the organizational specification is translated into a *normative program*, written in a specific language, called NOPL (Normative Organisation Programming Language) (Hübner et al., 2009; Hübner et al., 2010a; Hübner et al., 2011). The interpretation of such program is performed by a dedicated interpreter, included in each organizational artifact, and regulates the functioning of the organization.

A normative program in NOPL is composed of:

1. A set of normative facts, either translated from the specification or added dynamically during the execution;
2. A set of inference rules;
3. A set of norms.

Normative facts and inference rules follow a syntax similar to the one used in Jason and Prolog. Norms, in turn, have the form:

$$id : \phi \rightarrow \psi$$

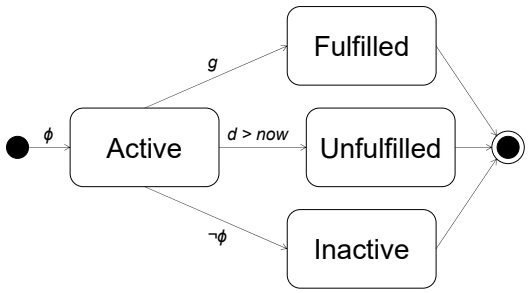

Figure 5.3: State transitions for obligations in JaCaMo, from (Hübner et al., 2009).

where id is a unique identifier, ϕ is a logical formula denoting the *activation condition* for the norm, and ψ is the *consequence* of the norm activation. ψ can be either the failure of the action triggering the norm, denoting the regimentation of a prohibition, or the emission of an obligation directed towards an agent and concerning a state of the world that the agent ought to bring about.

Obligations have a well-defined lifecycle, as reported in Figure 5.3. Once created, when the activation condition ϕ holds, an obligation is *active*. It becomes *fulfilled* when the agent, to which the obligation is directed, brings about the state of the world specified by the obligation (e.g., the achievement of a goal g) before a given deadline d. It is *unfulfilled* when the agent does not bring it about before the deadline. Should ϕ not hold anymore, the obligation becomes *inactive*.

The translation of an organizational specification results in multiple NOPL programs: a specific normative program is produced for each group or scheme and interpreted in the corresponding artifact. Program generation follows some translation rules (t-rules), as specified in (Hübner et al., 2011). For instance, for each goal in the scheme, a corresponding normative fact is included in the normative program:

$$\texttt{goal(}m\texttt{,}g\texttt{,}precond\texttt{,}type\texttt{,}\#achieve\texttt{,}ttf\texttt{)}$$

The fact defines a goal by specifying the missions m it belongs to, the goal identifier g, its preconditions $precond$, the type of goal, the number of agents having to achieve it, and its deadline ttf.

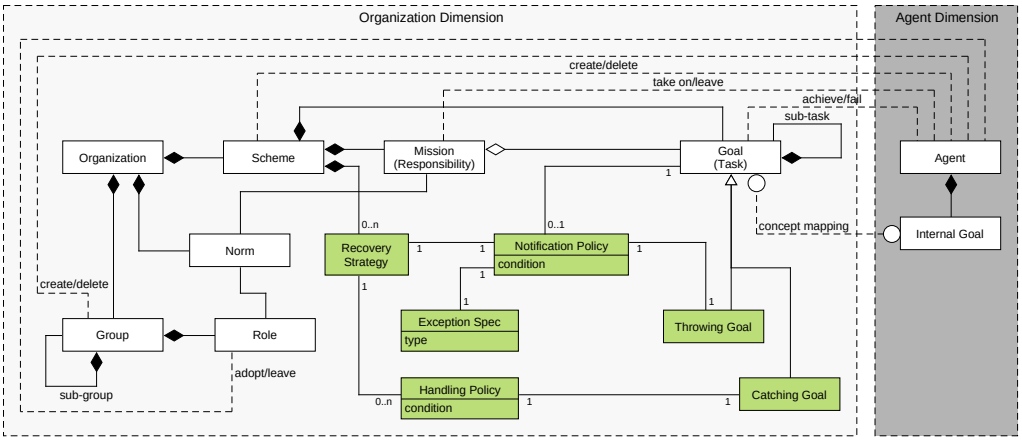

Figure 5.4: $\mathcal{M}$OISE's organizational metamodel extended for exception handling.

Besides facts, rules allow to infer the state of the scheme and of its goals. For instance, the following rule states that a goal is enabled (i.e., ready to be pursued) as soon as all its dependencies have been satisfied.

```
1 enabled(S,G) :- goal(_, G, dep(and,PCG), _, NP, _) &
2                 NP \== 0 \& all_satisfied(S,PCG).
```

Finally, norms are defined to ensure some properties, such as to avoid that more agents than the permitted ones commit to a mission and to issue obligations regarding the achievement of organizational goals.

5.2 Adding Exceptions[6]

With reference to the abstract model presented in the previous chapter, Figure 5.4 shows how it can be easily mapped to the concepts constituting $\mathcal{M}$OISE's organizational metamodel. Indeed, we can map JaCaMo's concepts of `Mission` and `Goal` respectively to the concepts of `Responsibility` and `Task` of our proposed abstract model. Similarly, here we interpret the agent commitment to a mission as an assumption of responsibility with respect to the mission goals.

[6]The full code of $\mathcal{M}$OISE extended with exception handling is available at `http://di.unito.it/moiseexceptions`.

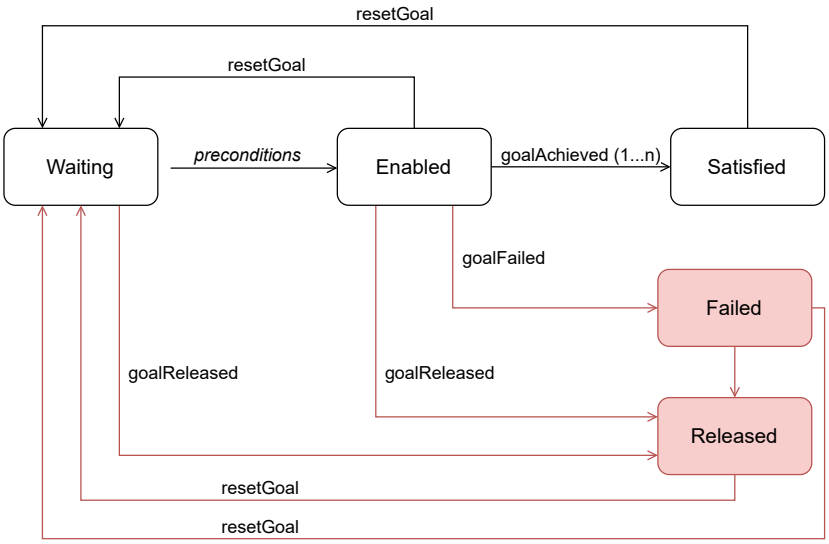

Figure 5.5: Extended lifecycle of a JaCaMo goal.

Let us recall our definition of exception within the scope of an organization: i.e., an event which denotes the impossibility, for some agent, to fulfill one of its responsibilities. We can cast it into JaCaMo's context by interpreting it as:

> *The impossibility for some agent to achieve a goal belonging to one of the missions it committed to, when requested by the organizational infrastructure through an obligation.*

Under this perspective, an exception could amount to an obligation unfulfillment, expressing the impossibility to achieve the corresponding goal before its deadline. Similarly, it could be an organizational event explicitly triggered by the agent at hand to notify a failure in the achievement of the assigned goal.

To model this aspect, we extended the lifecycle of a goal instance with two additional states: *failed* and *released*, as shown in Figure 5.5. A goal failure is eventually signaled by the agent responsible for it, similarly to goal achievement, denoting the impossibility for the agent to achieve the goal. Conversely, an agent can release a goal, meaning that the goal had not been achieved as planned, but some corrective actions have been undertaken and the execution of the scheme can proceed (as if the goal were satisfied).

5.2.1 Using Exceptions in the Organizational Specification

In accordance with the general model, for each exception deemed to possibly occur during the execution of a scheme, we enriched the scheme specification with the following concepts:

Recovery Strategy encodes when and how a given exception is to be raised and handled within the organization. Its role is to relate the raising of an exception to the agent in charge of handling that exception. As in the abstract model, it includes a notification policy and one or more handling policies.

Notification Policy specifies when the exception must be raised. It is characterized by a condition denoting the state of the organization management infrastructure corresponding to the exceptional situation. It is also associated with a goal representing the object of the exception (i.e., the goal that could not be completed), and with a throwing goal, to be enabled when such a circumstance hold.

Throwing Goal denotes the organizational goal of raising the exception, i.e., it will make the agent that is responsible for it to provide the information that is needed for recovery.

Exception Spec encodes the kind of feedback to be produced by the agent raising the exception, namely a set of ground first-order facts.

Handling Policy specifies a way in which the exception must be handled, once the needed information is available. It is characterized by a condition expressing the state of the organization management infrastructure in which the policy is applicable and it is associated with a catching goal.

Catching Goal captures the course of action to handle the exception and remediate. The aim of its achievement is to restore the normal execution of the scheme after an exception is raised.

Throwing goals and catching goals specialize the goal specification and are incorporated into mission just like standard ones. As a result, missions become a tool to

distribute the responsibilities, not only concerning the normal execution, but also for the management of exceptional situations. Policies, in turn, delimit the scope of such responsibilities, specifying when and how they are to be discharged.

The purpose of throwing goals, in particular is to make agents produce the relevant feedback concerning the reasons of the failure in case of exceptions. Catching goals, in turn, allow to involve in the recovery the right agents entitled for it.

Let us recall again Example 1. We can now extend the specification of the *ATM* scheme, in Listing 5.1, with the following recovery strategy targeting a `not a number` exception, possibly arising from the failure of goal `parseAmount`.

```
1  <recovery-strategy id="...">
2      <notification-policy id="...">
3          <condition type="goal-failure">
4              <condition-argument id="target" value="parseAmount" />
5          </condition>
6          <exception-spec id="nan">
7              <exception-argument id="index" arity="1" />
8          </exception-spec>
9          <goal id="throwNan" />           <!-- THROWING GOAL -->
10     </notification-policy>
11     <handling-policy id="...">
12         <condition type="always" />
13         <goal id="recoverFromNan" />     <!-- CATCHING GOAL -->
14     </handling-policy>
15 </recovery-strategy>
```

Listing 5.3: Recovery Strategy for a `not a number` exception in the *ATM* organizational specification.

Both the notification policy and the handling policy constituting the recovery strategy encompass a condition, denoting when the given policy is applicable. Each condition is characterized by a type and a (possibly empty) list of arguments, depending on the condition type. For notification policies, the condition expresses the perturbation being handled (i.e., the `must-notify-when` attribute in the abstract model). In this case, the condition at Lines 3-5 expresses that the policy is to be applied when a failure occurs in the achievement of goal `parseAmount`. As a consequence, the throwing goal `throwNan` (Line 9) is enabled and the agent responsible for it is required to throw the exception, i.e., provide a feedback complying with the exception spec included in the policy. Each exception spec is characterized by a

Type	Arguments	Condition formula
always	[]	true
goal-failure	[target]	scheme_id(S) & failed(S,$target)
goal-ttf-expiration	[target]	scheme_id(S) & unfulfilled(obligation(_,_,done(S,$target,_),_))
custom	[formula]	$formula

Table 5.1: Condition types for recovery strategies.

(possibly empty) list of arguments, specifying first order predicates (together with their arity) to be instantiated while throwing the exception. In this case (see Lines 6-8) the exception `nan` (not a number) must be thrown by specifying an index (e.g., the index of the first non numerical character found).

Similarly, the condition in a handling policy specifies when such policy is applicable. We recall that a recovery strategy could encompass more than one handling policies, each one applicable in different circumstances. The condition allows to determine when the corresponding catching goal (Line 13) must be enabled. In this case the condition type `always` states that the policy is applicable in any circumstance, as soon as the related exception has been thrown.

Condition types and arguments are mapped to logical formulas in the normative program (see Section 5.3 for a detailed discussion). Table 5.1 summarizes the condition types we have identified, along with their arguments and the corresponding formulas in NOPL[7]. In the logical formulas, placeholders starting with $ are substituted with the actual value specified for the corresponding argument. Custom conditions can be expressed as well, by directly specifying the corresponding formula as an argument.

5.2.2 Using Exceptions in Jason Agent Programming

The proposed exception handling architecture, whose implementation will be discussed in detail in Section 5.3, is seamlessly integrated into the organization management infrastructure. For this reason, agent programming in standard JaCaMo and in

[7] Condition types are stored in a dedicated configuration file. If needed, new condition types can be easily added by specifying the condition type, arguments and the corresponding forumla in NOPL.

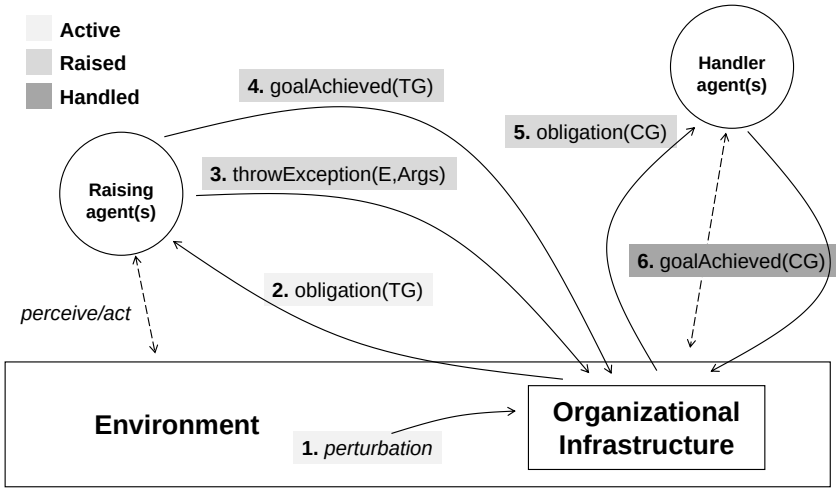

Figure 5.6: Interaction between agents and organization for exception handling.

our proposed extension follows a uniform approach. Agents, when entering into an organization, take on some responsibilities by committing to missions. On the basis of these responsibilities and of the state of the organization, the infrastructure issues obligations to achieve organizational goals, thereby coordinating the distributed execution of the scheme.

Agent developers are then required to implement the set of plans to make their agents able to fulfill their responsibilities, achieving the goals expressed by the obligations directed to them. With the exception handling mechanism put in place, agents will also have the possibility to notify the impossibility to achieve some of these goals, resulting in the occurrence of some exceptions.

Figure 5.6 shows the typical interaction schema, between the involved agents and the organizational infrastructure, for handling the occurrence of an exception. Recalling the diagram in Figure 4.3, the different steps are highlighted in shades of grey denoting how each JaCaMo-specific one contributes to the evolution of the general lifecycle status of the exception at hand (from active, to raised, to handled). As soon as a perturbation is detected, a suitable recovery strategy is searched and the corresponding notification policy is activated. An obligation to achieve the specified throwing goal `TG` is then issued to the agent responsible for it. To fulfill its responsibility, the agent will have to throw the corresponding exception, providing

appropriate exception arguments. The exception is considered as fully raised once the throwing goal `TG` is marked as achieved. This enables one (or more) handling policy and the obligation(s) for the related catching goal(s) `CG` is issued. Again, the exception is treated as handled only once the catching goal(s) `CG` is marked as achieved.

Listing 5.4 shows the implementation of the *parser* agent, responsible for goal `parseAmount` in the *ATM* organization, extended for exception handling.

```
1  +obligation(Ag,_,done(_,parseAmount,Ag),_)
2       : .my_name(Ag)
3      <- !parseAmount;
4         goalAchieved(parseAmount).
5
6  +!parseAmount
7      <- parseAmount. //Operation over an environment artifact
8
9  -!parseAmount
10     <- goalFailed(parseAmount);
11        .fail.
12
13 +!throwNan
14      : //Retrieve index I of the first non-numeric character
15     <- throwException(nan,[index(I)]).
```

Listing 5.4: *Parser* agent in the *ATM* organization, extended for exception handling.

We can extend the agent's code with a contingency plan (see Section 3.9.1), at Line 9, to be triggered should parsing fail. The failure, which at this point is local to the agent, causes the impossibility to achieve the organizational goal as well. The agent has then the possibility to inform the organizational infrastructure by executing the `goalFailed(...)` operation at Line 10.

The agent is also equipped with an additional plan, at Line 13, triggered as soon as the agent receives an obligation for the organizational goal `throwNan`[8]. This goal is the throwing goal included in the notification policy specified in Listing 5.3, targeting right the failure of `parseAmount`. In other words, the agent is responsible not only for performing the actual parsing, but also to provide a feedback, should an

[8] We recall that in JaCaMo a library of predefined plans allows to map obligations to agents' internal goals automatically. For this reason in some cases we might have, as a plan triggering event, directly the corresponding internal goal rather than the issued obligation. The library also marks the organizational goal as achieved as soon as the internal goal is achieved.

exception occur in the execution of its assigned task. To fulfill this responsibility, the agent must throw a `nan` exception and provide the index of the first non-numeric character identified (see Line 15). The primitive `throwException(...)` serves this purpose. More details about the operations provided by the organizational infrastructure to support exception handling can be found in Section 5.3.3.

Listing 5.5 below, in turn, reports the code of the *request handler* agent, which is responsible for handling the exception thrown by the *parser*.

```
1  attempts(1).
2
3  +obligation(Ag,_,done(_,recoverFromNan,Ag),_)
4       : .my_name(Ag) &
5         attempts(N) & N < 3
6      <- -attempts(N);
7         +attempts(N+1);
8         resetGoal(obtainAmount).
9
10 +obligation(Ag,_,done(_,recoverFromNan,Ag),_)
11      : .my_name(Ag) &
12        attempts(N) & N >= 3
13     <- goalFailed(recoverFromNan).
14
15 +obligation(Ag,_,done(_,throwAmountUnavailable,Ag),_)
16      : .my_name(Ag)
17     <- throwException(amountUnavailable,[errorCode(...)]);
18        goalAchieved(throwAmountUnavailable).
```

Listing 5.5: *Request handler* agent in the *ATM* organization.

The recovery strategy specifies a handling policy encompassing a catching goal `recoverFromNan`. As soon as the `nan` exception is thrown by the *parser*, such a goal becomes enabled and the corresponding obligation issued to the *request handler*. It's worth noting that the agent remains completely free to autonomously deliberate the best course of actions in order to concretely deal with the exceptional situation. The rationale is that, by virtue of its taken responsibilities, the agent should be the one deemed to have the right expertise to solve (or at least mitigate) the problem.

In this case, the agent is equipped with two alternative plans (Lines 3 and 10). The approach adopted by the agent to deal with the `nan` exception is to make the user insert another amount, up to three times. To this end, the agent keeps track of

the number of attempts made so far (Line 1). If the number of attempts is less than three, the first plan is triggered and the exception is handled by resetting goal `obtainAmount`.

As already pointed out, it's worth noting that throwing and catching goals may fail, as well, causing the occurrence of further exceptions. This is exactly what happens after three unsuccessful parsing attempts. In that case, the obligation to pursue the catching goal `recoverFromNan` triggers the second plan, at Line 10, leading to the failure of the goal itself (see Line 13). To cope with the exception, an additional recovery strategy can be added to the functional specification of the organization, as reported in Listing 5.6, below.

```
1  <recovery-strategy id="...">
2      <notification-policy id="...">
3          <condition type="goal-failure">
4              <condition-argument id="target" value="recoverFromNan" />
5          </condition>
6          <exception-spec id="amountUnavailable">
7              <exception-argument id="errorCode" arity="1" />
8          </exception-spec>
9          <goal id="throwAmountUnavailable" />
10     </notification-policy>
11     <handling-policy id="...">
12         <condition type="always" />
13         <goal id="retryLater" />
14     </handling-policy>
15 </recovery-strategy>
```

Listing 5.6: Recovery stategy targeting the `amount unavailable` exception.

The recovery strategy encompasses a notification policy targeting the failure of goal `recoverFromNan`. An exception spec `amountUnavailable` is defined including an argument called `errorCode` (see Lines 6-8).

Let us assume that the responsibility for the throwing goal is taken by the *request handler* agent, too, and that the responsibility for the catching goal is taken by an *ATM handler* agent, in charge of supervising the functioning of the ATM as a whole. As soon as `recoverFromNan` is marked as failed, goal `throwAmountUnavailable` is enabled and assigned to the *request handler* for achievement, triggering the plan at

Line 15 in Listing 5.5. The agent then fulfills the obligation by concretely throwing the exception and specifying the needed error code (Line 17).

Finally, Listing 5.7 shows an excerpt of the *ATM handler* agent's code.

```
1 +!retryLater
2      : exceptionArgument(amountUnavailable, errorCode(...))
3     <- closeSession;
4        goalReleased(withdraw).
```

Listing 5.7: *ATM handler* agent in the *ATM* organization.

The plan above allows the agent to fulfill the responsibility concerning catching goal `retryLater`. In this particular case, the exception is handled by closing the session with the user and by releasing the root goal of the scheme, i.e., `withdraw`. By doing that, the agent notifies that the initial organizational goal won't be achieved anymore, and the execution of the scheme is stopped gracefully.

The belief in the plan context (Line 2) amounts to an observable property of the organizational artifacts encoding the argument specified by the *request handler* while throwing the exception, i.e., the error code in this case. It's worth noting that multiple plans could be defined, encompassing different courses of actions to deal with the exception, depending on the error code at hand.

5.3 Implementation

We now illustrate how the $\mathcal{M}$OISE's infrastructure has been concretely extended to support exception handling, as shown above.

5.3.1 Extending the Specification's XML Schema

As said, in JaCaMo, organizational specifications are written in XML. To realize the picture described in Section 5.2.1, we extended the XML schema that formally describes how $\mathcal{M}$OISE organizations must be specified.

In particular, we extended the `scheme` element definition so that any scheme can encompass a variable number of recovery strategies, as follows.

```
<xsd:element maxOccurs="unbounded" minOccurs="0" name="scheme">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element maxOccurs="1" minOccurs="0"
               name="properties" type="moise:propertiesType"/>
            <xsd:element maxOccurs="1" minOccurs="1"
               name="goal" type="moise:goalDefType"/>
            <xsd:element maxOccurs="unbounded" minOccurs="0"
               name="recovery-strategy"
               type="moise:recoveryStrategyType"/>
            <xsd:element maxOccurs="unbounded" minOccurs="0"
               name="mission" type="moise:missionType"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string"/>
    </xsd:complexType>
</xsd:element>
```

Listing 5.8: `scheme` element extended in $\mathcal{M}$OISE's XML schema.

Besides goals and missions, a scheme can include zero-to-many recovery strategy elements, identified by the tag `recovery-strategy` (Lines 8-10).

Furthermore, we introduced the following new element types, describing the concepts related to exception handling.

```
<xsd:complexType name="recoveryStrategyType">
    <xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="1"
           name="notification-policy"
           type="moise:notificationPolicyType"/>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
           name="handling-policy"
           type="moise:handlingPolicyType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>
```

Listing 5.9: Element type encoding a recovery strategy in $\mathcal{M}$OISE's XML schema.

Each `recovery-strategy` element is characterized by an `id` and must include exactly one notification policy (identified by the tag `notification-policy`) and zero-to-many handling policies (tag `handling-policy`).

```
<xsd:complexType name="notificationPolicyType">
    <xsd:sequence>
```

```
        <xsd:element maxOccurs="1" minOccurs="0"
           name="properties" type="moise:propertiesType"/>
        <xsd:element maxOccurs="1" minOccurs="1"
           name="condition" type="moise:conditionType"/>
        <xsd:element maxOccurs="1" minOccurs="1"
           name="exception-spec" type="moise:exceptionSpec"/>
        <xsd:element maxOccurs="1" minOccurs="1"
           name="goal" type="moise:goalDefType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>
```

Listing 5.10: Element type encoding a notification policy in $\mathcal{M}$OISE's XML schema.

Each `notification-policy` element has an `id` and includes a `condition`, expressing the activation condition of the policy, an `exception-spec` element and a `goal` element (i.e., the throwing goal).

```
<xsd:complexType name="conditionType">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
            name="condition-argument"
            type="moise:conditionArgumentType"/>
    </xsd:sequence>
    <xsd:attribute name="type" type="xsd:string" use="required" />
</xsd:complexType>
<xsd:complexType name="conditionArgumentType">
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required" />
</xsd:complexType>
```

Listing 5.11: Element type encoding a policy condition in $\mathcal{M}$OISE's XML schema.

`condition` elements allow to specify the activation conditions of notification and handling policies. Each `condition` has a type (see Table 5.1) and encompasses a (possibly empty) set of `condition-arguments`. A `condition-argument` is a key-value pair.

```
<xsd:complexType name="exceptionSpec">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" minOccurs="0"
           name="exception-argument"
           type="moise:exceptionArgumentType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" use="required" />
</xsd:complexType>
<xsd:complexType name="exceptionArgumentType">
    <xsd:attribute name="id" type="xsd:string" use="required"/>
```

```
    <xsd:attribute name="arity" type="xsd:integer" use="required" />
</xsd:complexType>
```

Listing 5.12: Element type encoding an exception spec in $\mathcal{M}$OISE's XML schema.

Similarly, `exception-spec` elements encode exception specifications. Each element of this type has an `id` and a (possibly empty) set of exception arguments. Each argument is characterized by an `id` and an `arity`. `id` and `arity` specify the functor name and the arity of one of the predicates that will have to be instantiated by the agents while raising an exception compliant with the exception spec at hand.

```
<xsd:complexType name="handlingPolicyType">
    <xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="0"
           name="properties" type="moise:propertiesType" />
        <xsd:element maxOccurs="1" minOccurs="1"
           name="condition" type="moise:conditionType" />
        <xsd:element maxOccurs="1" minOccurs="1"
           name="goal" type="moise:goalDefType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string" use="required" />
</xsd:complexType>
```

Listing 5.13: Element type encoding a handling policy in $\mathcal{M}$OISE's XML schema.

Finally, as before, each `handling-policy` element has an `id`. It includes an activation `condition` and a `goal` element (the catching goal).

5.3.2 Extending the Normative Program

At runtime, the XML specification of a JaCaMo organization is then translated into a set of normative NOPL programs, which address groups and schemes. We extended the normative program resulting from the translation of the scheme specification with the concepts of `Recovery Strategy`, `Notification Policy`, `Throwing Goal`, `Exception Spec`, `Handling Policy`, and `Catching Goal`. We now explain the facts, rules and norms that result from the translation of recovery strategy specifications, which together constitute the core of our exception handling mechanism.

Facts. For each scheme, we enrich the normative program with the following normative facts.

`recoveryStrategy(RS)` denoting that the scheme includes a recovery strategy with id `RS`.

`notificationPolicy(NP,Condition)` encoding a notification policy with id `NP`. `Condition` is the logical formula mapped from the policy condition, as described in Table 5.1.

`handlingPolicy(HP,Condition)` for an handling policy with id `HP` and condition `Condition`.

`strategy_policy(RS,P)` encoding that a given policy `P` belongs to a recovery strategy `RS`.

`policy_goal(P,G)` specifying the relation between a goal `G` and the policy `P` it belongs to. Depending on the kind of policy (either a notification or a handling one) the goal will be a throwing goal or a catching goal.

`exceptionSpec(E)` for an exception specification with id `E`.

`policy_exceptionSpec(NP,E)` denoting that the exception spec `E` is defined within the scope of notification policy `NP`.

`exceptionArgument(E,ArgFunctor,ArgArity)` denoting that the exception spec `E` encompasses an argument, which consists of a first-order predicate with functor `ArgFunctor` and arity `ArgArity`. Multiple arguments can be associated with a given exception spec.

Moreover, the following normative facts may be dynamically added during the execution of the scheme, as a result of the execution of specific operations over the SchemeBoard artifact (see Section 5.3.3). Dynamic facts, added or inferred at runtime, constitute the *normative state* of the organization.

`failed(S,G)` denoting that a failure occurred in the achievement of goal `G` in scheme `S`.

`released(S,G)` denoting that goal `G` has been released.

`thrown(S,E,Ag,Args)` denoting that an exception has been thrown by agent `Ag`, following the exception spec `E`. `Args` is a list of arguments, i.e., a set of ground predicates having the same structures of the arguments specified with `exceptionArgument(E,ArgFunctor,ArgArity)` for exception `E`.

The following excerpt of code shows the normative facts resulting from the translation of the recovery strategy specified in Listing 5.3.

```
recoveryStrategy(rec1).

notificationPolicy(np1,(scheme_id(S) & failed(S,parseAmount))).

handlingPolicy(hp1,true).

strategy_policy(rec1,np1).
strategy_policy(rec1,hp1).

exceptionSpec(nan).

exceptionArgument(nan,index(Arg0)).

policy_exceptionSpec(np1,nan).

policy_goal(np1,throwNan).
policy_goal(hp1,recoverFromNan).
```

Listing 5.14: Recovery strategy for `not a number` translated in NOPL.

Whilst normative facts are deduced specifically on the basis each functional specification or added dynamically, rules and norms are general, and applicable to any scheme instance.

Rules. Rules, in particular, allow to define when to enable throwing and catching goals on the basis of the policy they belong to.

For throwing goals we have defined the following rule.

```
enabled(S,TG) :-
    policy_goal(P,TG) &
    notificationPolicy(P,Condition) &
    Condition &
    goal(_, TG,  Dep, _, NP, _) & NP \== 0 &
```

```
6       ((Dep = dep(or,PCG)  & (any_satisfied(S,PCG) | all_released(S,PCG))) |
7        (Dep = dep(and,PCG) & all_satisfied_released(S,PCG))
8       ).
```

Listing 5.15: NOPL rule which enables throwing goals.

A throwing goal `TG` must be enabled as soon as the `Condition` defined for the policy it belongs to holds. Since a throwing goal can be complex, encompassing multiple sub-goals, its preconditions must be satisfied (or released) accordingly.

For catching goals the following rule applies.

```
1  enabled(S,CG) :-
2       policy_goal(HP,CG) &
3       handlingPolicy(HP,Condition) &
4       Condition &
5       recoveryStrategy(ST) &
6       strategy_policy(ST,HP) &
7       strategy_policy(ST,NPol) &
8       policy_exceptionSpec(NPol,E) &
9       thrown(S,E,_,_) &
10      policy_goal(NPol,TG) &
11      satisfied(S,TG) &
12      goal(_, CG,  Dep, _, NP, _) & NP \== 0 &
13      ((Dep = dep(or,PCG)  & (any_satisfied(S,PCG) | all_released(S,PCG))) |
14       (Dep = dep(and,PCG) & all_satisfied_released(S,PCG))
15      ).
```

Listing 5.16: NOPL rule which enables catching goals.

Similarly to throwing goals, a catching goal is enabled if the condition specified in the policy it belongs to holds and if the precondition goals are satisfied. However, for catching goals this is not enough. We additionally require that an exception has actually been thrown (see Line 9). Such exception must follow the specification given by the notification policy belonging to the same strategy (Lines 5-8). Moreover, the corresponding throwing goal must be satisfied (Line 11).

In this way, we ensure that the agent(s) that will be in charge of handling the exception, to whom the catching goal is assigned, will be able to take advantage of the information provided upon the exception throwing.

Norms. Agents are asked to pursue throwing and catching goals by means of the standard built-in norm for goal achievement, reported in Listing 5.17.

```
1  // agents are obliged to fulfill their enabled goals
2  norm ngoal:
3     committed(A,M,S) & mission_goal(M,G) &
4     ((goal(_,G,_,achievement,_,D) & What = satisfied(S,G)) |
5      (goal(_,G,_,performance,_,D) & What = done(S,G,A))) &
6     well_formed(S) &
7     not satisfied(S,G) &
8     not failed(_,G) &
9     not released(_,G) &
10    not super_satisfied(S,G)
11 -> obligation(A,(enabled(S,G) & not failed(S,G)),What,`now` + D).
```

Listing 5.17: NOPL norm issuing obligations to achieve goals.

We slightly modified the norm in order to avoid the issuing of obligations concerning goals marked as failed or released (Lines 8-9).

We then added some regimented norms to ensure some properties. First of all, we want to avoid that agents can signal as failed goals that are not enabled, yet. The following norm serves the purpose.

```
1  norm fail_not_enabled_goal:
2     failed(S,G) &
3     mission_goal(M,G) &
4     not mission_accomplished(S,M) &
5     not enabled(S,G)
6  -> fail(fail_not_enabled_goal(S,G)).
```

Listing 5.18: NOPL norm regimenting goal failure.

The following norm, in turn, ensures that exceptions can be thrown only by following a well-defined exception spec.

```
1  norm exc_unknown:
2     thrown(S,E,Ag,Args) &
3     not exceptionSpec(E)
4  -> fail(exc_unknown(S,E,Ag)).
```

Listing 5.19: NOPL norm regimenting the throwing of unknown exceptions.

We also would like exceptions to be thrown only when a perturbation actually occurs and not arbitrarily by the agents. In other words, we must ensure that an exception can only be thrown if the condition of the corresponding notification policy holds.

```
1 norm exc_condition_not_holding:
2    thrown(S,E,Ag,Args) &
3    exceptionSpec(E) &
4    policy_exceptionSpec(NP,E) &
5    notificationPolicy(NP,Condition) &
6    policy_goal(NP,TG) &
7    not (Condition | satisfied(S,TG))
8 -> fail(exc_condition_not_holding(S,E,Ag,Condition)).
```

Listing 5.20: NOPL norm regimenting exception throwing conditions.

It's worth noting that we allow the condition not to hold if the throwing goal has been already satisfied (Line 7). The rationale is that, after a successful handling of the exception the critical condition will likely stop holding. Nonetheless the fact `thrown(S,E,Ag,Args)`, together with `satisfied(S,TG)`, keeps track of the fact that an exceptional situation occurred (and has been handled).

Thanks to missions, a developer can identify at design time the classes of agents (i.e., their roles) which will be responsible for throwing exceptions, when needed. In this way, we ensure that the agents will likely be provided with the right capabilities (expertise, opportunity, knowledge, etc.) and powers, within the organizational scope, to provide the needed feedback. For the very same reason, it is important to ensure that only these designated agents can throw exceptions. To this end we defined the following norm.

```
1 norm exc_agent_not_allowed:
2    thrown(S,E,Ag,Args) &
3    exceptionSpec(E) &
4    mission_goal(M,TG) &
5    policy_exceptionSpec(NP,E) &
6    policy_goal(NP,TG) &
7    not committed(Ag,M,S)
8 -> fail(exc_agent_not_allowed(S,E,Ag)).
```

Listing 5.21: NOPL norm regulating agents allowed to throw exceptions.

The norm inhibits the throwing of exceptions by agents not committed to the mission encompassing the corresponding throwing goal.

At the same time, the entitled agents, should actually provide the requested information to fulfill their responsibility. In other words, a throwing goal can be

marked as achieved only if the corresponding exception has actually been thrown beforehand.

```
1 norm ach_thr_goal_exc_not_thrown:
2    done(S,TG,Ag,Args) &
3    exceptionSpec(E) &
4    policy_exceptionSpec(NP,E) &
5    policy_goal(NP,TG) &
6    not super_goal(SG,TG) &
7    not thrown(S,E,_,_)
8 -> fail(ach_thr_goal_exc_not_thrown(S,G,E,Ag)).
```

Listing 5.22: NOPL norm regulating the achievement of throwing goals.

Since throwing goals can be complex, encompassing multiple sub-goals, the norm requires the exception to be thrown before the achievement of the root goal (see Line 6).

Finally, the last three norms allow to ensure that the arguments provided by the agents while throwing an exception follow the specification. More specifically, the following norm ensures that the arguments of a `thrown(S,E,Ag,Args)` predicate are ground predicates, i.e., they don't contain variables.

```
1 norm exc_arg_not_ground:
2    thrown(S,E,Ag,Args) &
3    exceptionSpec(E) &
4    .member(Arg,Args) &
5    not .ground(Arg)
6 -> fail(exc_arg_not_ground(S,E,Arg)).
```

Listing 5.23: NOPL norm regimenting exception arguments groundness.

At the same time, it is important to ensure that all the relevant information is provided, i.e., that all the required arguments are instantiated.

```
1 norm exc_arg_missing:
2    thrown(S,E,Ag,Args) &
3    exceptionSpec(E) &
4    exceptionArgument(E,ArgFunctor,ArgArity) &
5    not (.member(Arg,Args) &
6         Arg=..[ArgFunctor,T,A] &
7         .length(T,ArgArity)
8    )
9 -> fail(exc_arg_missing(S,E,ArgFunctor,ArgArity)).
```

Listing 5.24: NOPL norm regulating the absence of required exception arguments.

The NOPL construct `P=..[F,T,A]`, used at Line 6, allows to determine the functor `F`, terms `T`, and eventually annotations `A`, of a predicate `P`. The norm triggers a failure if at least one of the arguments specified for `E` does not unify with one of the terms in the list `Args` of dynamic fact `thrown(S,E,Ag,Args)`. Indeed, this means that some of the information the agent was requested to provide is still missing.

The last norm regiments the prohibition, for agents, to include in the throwing of an exception with undesired arguments.

```
1 norm exc_arg_unknown:
2    thrown(S,E,Ag,Args) &
3    exceptionSpec(E) &
4    .member(Arg,Args) &
5    Arg=..[ArgFunctor,T,A] &
6    .length(T,ArgArity) &
7    not exceptionArgument(E,ArgFunctor,ArgArity)
8 -> fail(exc_arg_unknown(S,E,Arg)).
```

Listing 5.25: NOPL norm prohibiting undesired exception arguments.

The norm is triggered if one of the arguments in `Args` does not follow the argument specification for `E`. The result is a failure in the exception throwing action.

5.3.3 Extending the Organizational Artifacts

As already explained, in JaCaMo the organization management infrastructure is realized through a set of artifacts. Such artifacts allow agents to interact with the organization, by perceiving its observable state and by executing some operations, such as "commit to mission" or "goal x has been achieved" (Hübner et al., 2009). JaCaMo's NOPL interpreter relies on these artifacts, so, in order to introduce in JaCaMo the exception handling mechanism explained above, it has been necessary to extend one of them. Precisely, we enriched the SchemeBoard artifact for scheme management with three additional operations:

`goalFailed(G)` to set an organizational goal `G` as failed. It adds to the normative state the dynamic fact `failed(S,G)`, where `S` is the scheme identifier.

`throwException(E,Args)` to throw an exception `E` with a list of arguments `Args`. It adds to the normative state the fact `thrown(S,E,Ag,Args)`, where `Ag` is the name of the agent executing the operation.

`goalReleased(G)` to release an organizational goal `G`. It adds to the normative state the fact `released(S,G)`.

The first operation, `goalFailed(G)`, is to be used by the agents to proactively signal the occurrence of a perturbation in the functioning of the organization, i.e., a failure in the agents' fulfillment of their responsibilities. Such a perturbation will trigger the exception handling mechanism, possibly enabling a throwing goal on the basis of the specified recovery strategies.

`throwException(E,Args)`, in turn, allows the agents in charge of throwing the exceptions to fulfill their responsibility and provide a feedback about the context where exceptions occur. The exception arguments are made available to agents as artifact's observable properties having the shape `exceptionArgument(E,Arg)` (one property for each argument).

Finally, `goalReleased(G)` serves as a means, available to the appointed agents, for notifying to the organization that an exception has been handled. It's worth noting that releasing a goal is not the only way to handle a given exception. Resetting the goal, for instance, could be an alternative, as well. This could be the result of the execution of some actions restoring the possibility to achieve a failed goal. Under this perspective, we let to the agents responsible for handling the choice of the best actions to deal with exceptions.

6 Experimentation and Evaluation

Contents

In this chapter we illustrate, evaluate and discuss the main features of the proposed exception handling approach and infrastructure by means of a set of practical use cases. They take inspiration from real-world scenarios and practices – coming from

the fields, e.g., of house constructions, business processes, smart factories, etc. – and are modeled as multi-agent organizations.

To this end we show excerpts of implementations that leverage the version of JaCaMo extended with the exception handling mechanism presented in the previous chapter. In particular, the examples will help to evaluate the proposal w.r.t. to the following features that, we believe, are important for any exception handling mechanism to be suitable for MAS: *autonomy preservation*, *decentralization*, *responsibility distribution*, *availability of reliable feedback*, and *platform integration*. More details are given in Section 6.6.

In particular, Section 6.1 reviews the main features of the proposed mechanism. The main benefits coming from its integration in the JaCaMo platform are highlighted. The proposal is then compared with an approach based on simple message passing to underline the need for an explicit responsibility distribution among the agents for exception handling. The scenario illustrated in Section 6.2 highlights the benefits coming from the availability of informed feedback to handle exceptions in a distributed and decentralized setting. At the same time, we discuss how the approach allows to preserve the agents' autonomy and how such an autonomy is an enabler itself for effective exception handling. In Section 6.3, we compare the solution with the exception handling mechanism adopted in BPMN (see Section 2.4). The main aim of the section is to emphasize that multi-agent systems have the potential to support the realization of complex business processes, but, to this end, they must be equipped with the means to address exceptions, as effectively achieved by the formalism. In Section 6.4, exception handling is put in place in an industrial scenario where a production cell is modeled as a MAS. The scenario points out how our proposal leverages the distributed and decentralized nature of agents in the exception handling process, as well. Finally, the example presented in Section 6.5 shows how the proposed approach enables dynamic improvements in the functioning of an organization. Thanks to exception handling, agents can effectively adapt to adverse contextual conditions, with straightforward benefits for the whole organization.

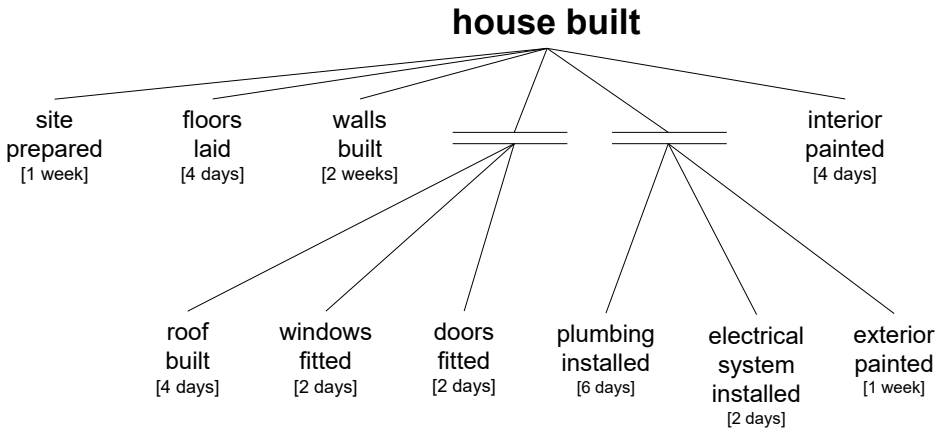

Figure 6.1: Functional decomposition of the organizational goal in the *building-a-house* organization, as specified in (Boissier et al., 2013).

6.1 Feature Overview: a Robust House Building

As a first practical scenario, to review the main features of the mechanism, we rely on the *building-a-house* example, originally introduced in (Boissier et al., 2013). Here, an agent wants to build a house on a plot. To achieve the goal the companies, it has contracted with, must coordinate and execute various tasks. The deployment of an organization thus serves the purpose effectively and Figure 6.1 graphically depicts a possible functional decomposition of the organizational goal. Sub-goals must be achieved in sequence, from left to right. Sub-goals grouped under a double horizontal line can be achieved in parallel. The structural specification, in turn, defines a group which includes the following roles: *house owner*, *site prep contractor*, *bricklayer*, *roofer*, *windows fitter*, *door fitter*, *plumber*, *electrician*, and *painter*. The *house owner* agent will be in charge of the overall house construction, i.e., of the root goal `house_built`, whilst each involved company, adopting the suitable role, will be responsible for a leaf goal in the decomposition tree.

A construction scenario, like the one described above, is a very dynamic environment where exceptional situations are likely to occur. As a consequence, robustness of the organization coordinating the building of the house is important. Achievement of the organizational goal involves the coordination of multiple companies executing the various sub-goals, part of which can be executed in parallel, while part depends

on others. Site preparation, e.g., must be completed before any other step. At the same time, agents may fail to discharge their responsibilities for a wide number of reasons and such failures could impact the organization as a whole. For instance, should the *site prep contractor* agent in charge of `site_prepared` face a failure, the whole house construction could not proceed. Depending on the reasons for a failure, different corrective actions might have to be taken, as well.

Let us focus, in particular, on two kinds of exceptions which may occur: a failure of goal `site_prepared` and a delay in the achievement of `windows_fitted`. The former is proactively caused by the responsible agent by notifying the failure, while the latter occurs as soon as the goal deadline for achievement is not respected.

6.1.1 Handling Goal Failure Exceptions

For what concerns the first exception, we can extend the functional specification of the organization with the following recovery strategy, targeting the failure in the achievement of `site_prepared`.

```
<recovery-strategy id="rsSitePreparation">
    <notification-policy id="npSitePreparation">
        <condition type="goal-failure">
            <condition-argument id="target" value="site_prepared" />
        </condition>
        <exception-spec id="site_preparation_exception">
            <exception-argument id="errorCode" arity="1" />
        </exception-spec>
        <goal id="notify_site_preparation_problem" />
    </notification-policy>
    <handling-policy id="hpSitePreparation">
        <condition type="always" />
        <goal id="handle_site_problem">
            <plan operator="parallel">
                <goal id="inspect_site" />
                <goal id="notify_affected_companies" />
            </plan>
        </goal>
    </handling-policy>
</recovery-strategy>
```

Listing 6.1: Recovery strategy targeting a failure in site preparation in the *building-a-house* scenario.

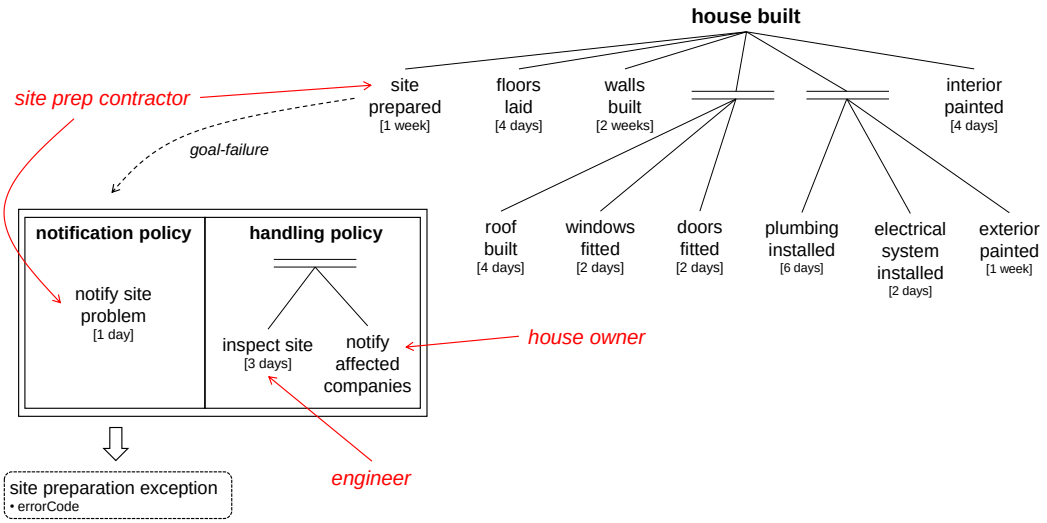

Figure 6.2: Functional decomposition of the *building-a-house* organizational goal extended with the recovery strategy targeting the failure of `site_prepared`.

Notification policy `npSitePreparation` specifies that, should a goal failure concerning `site_prepared` occur (see the condition at Lines 3-5), the throwing goal `notify_site_preparation_problem` is to be enabled. Its purpose is to make the agent responsible for it specify the reason for the failure, in order to exploit such information for recovery. To this end, an exception spec `site_preparation_exception` (Lines 6-8) specifying an `errorCode` argument is defined.

Handling policy `hpSitePreparation`, in turn, expresses what needs to be done to solve the site preparation exception, once it has been raised and the failure reason provided. In this case, the catching goal is composite (Lines 13-18): the site should be inspected and, at the same time, the other companies involved in the house construction notified. It's worth noting that agents in charge of these goal will have the possibility to leverage the information provided beforehand to achieve them. Site inspection, e.g., will be performed in different ways in case the failure is due to a flooding rather than to the finding of archaeological remains. Corrective actions undertaken by the agents to recover from the situation will be different, as well.

Figure 6.2 illustrates the functional decomposition of the *building-a-house* organizational scheme, extended with the recovery strategy presented above. Agents, that are responsible for some goals, are highlighted in red.

Specifically, *house owner* has now the responsibility for goal `notify_affected_companies`. An *engineer*, instead, is responsible for `inspect_site`: according to the raised exception, the result of the inspection, and its expertise, the agent will deliberate the most appropriate countermeasures. Finally, the *site prep contractor*, in charge of `site_prepared`, is also responsible for raising the exception by accomplishing goal `notify_site_preparation_problem`, when needed.

Having extended the organization specification with a recovery strategy for the eventual failure of `site_prepared`, we can now focus on agent programming. Listing 6.2 shows an excerpt of a possible implementation of the *site prep contractor* agent.

```
1  +obligation(Ag,_,done(_,site_prepared,Ag),_)
2       : .my_name(Ag)
3      <- !site_prepared;
4         goalAchieved(site_prepared).
5
6  +!site_prepared
7      <- prepareSite. //Simulate the action in the environment
8
9  -!site_prepared
10     <- goalFailed(site_prepared);
11        .fail.
12
13 +obligation(Ag,_,done(_,notify_site_preparation_problem,Ag),_)
14      : .my_name(Ag) &
15        //percepts encoding that the site is flooded
16     <- throwException(site_preparation_exception,[errorCode(flooding)]);
17        goalAchieved(notify_site_preparation_problem).
```

Listing 6.2: Code of the *site prep contractor* agent, raising the `site preparation exception`.

Notably, the agent discharges its responsibilities by way of two plans, reacting to two obligations. The first one (Line 1) refers to the achievement of goal `site_prepared`. The second one (Line 13) refers to the raising of an exception whenever goal `site_prepared` fails. In other words, the first obligation is issued in relation to the "standard" goal the agent is responsible for, whereas the second obligation is issued in relation to a recovery strategy, again under the responsibility of the agent. The obligation to achieve `site_prepared` is mapped onto an internal goal (Line 3). Should, for any reason, the agent fail to achieve such goal, the contingency plan at Line 9 would be triggered.

The execution of the `goalFailed` operation, at Line 10, allows the agent to notify the organization that something went wrong. The organization, in turn, activates the exception handling mechanism, according to the recovery strategy described above, by issuing an obligation to achieve `notify_site_preparation_problem` to the very same agent. This obligation requests the agent to raise an exception and to provide an error code, encoding the reason for the failure. The agent may be equipped with multiple plans to perform this task. The plan at Line 13 is activated when the reason of the failure amounts to a flooding. The exception is raised by the operation at Line 16. In particular, the second parameter of this operation is the list of ground predicates (i.e., the exception arguments), which must follow the structure specified by the exception spec in the recovery strategy. In this case, the agent includes predicate `errorCode` (of arity 1) with argument `flooding`. As already explained, the exception arguments encode the local knowledge that is deemed relevant to handle the exception, and that needs to flow from the agent, responsible for raising the exception (holder of such knowledge), to the agent responsible for handling the exception.

Having discussed how the exception is raised, we now consider how it is handled. In our simple recovery strategy, part of the handling is to be performed by the *engineer* agent. An excerpt of a possible implementation is shown in Listing 6.3, below.

```
1  +obligation(Ag,_,done(_,inspect_site,Ag),_)
2         .my_name(Ag) &
3         exceptionArgument(site_preparation_exception,
4                           failureReason(flooding))
5      <- performSiteAnalysis(Result);
6         fixFlooding(Result);
7         goalReleased(site_prepared);
8         goalAchieved(inspect_site).
9
10 +obligation(Ag,_,done(_,inspect_site,Ag),_)
11        .my_name(Ag) &
12        exceptionArgument(site_preparation_exception,
13                          failureReason(archaeologicalRemains))
14     <- delimitSite;
15        carefullyRemoveRemains;
16        resetGoal(site_prepared).
```

Listing 6.3: Code of the *engineer* agent in the *building-a-house* scenario.

One main advantage of our proposed exception handling mechanism is that it allows to conjugate two dimensions. On one hand, it gives the possibility to specify at an organizational level that some exceptional situations ought to be treated in order to ensure the right functioning of the organization. On the other hand, it preserves the agents' autonomy in choosing the best way to handle the exception, according to their know-how (i.e., capabilities and knowledge).

Also in this case, we focus on the plans the agent uses to discharge its responsibilities. Indeed, the agent in Listing 6.3 is equipped with two alternative plans to discharge its responsibility to handle the exception, depending on the failure reason specified together with the `site_preparation_exception`. The first plan, triggered when the error code denotes a flooding, encompasses the performing of a site analysis (e.g., to estimate the damages) and then some fixes. In this case, goal `site_prepared` is released (Line 7), so that after the fix the construction can proceed. If, instead, the second plan is triggered, denoting the presence of archaeological remains, the course of actions to undertake is different. In this case the site is firstly delimited, then the remains are carefully removed, and finally goal `site_prepared` is reset, so that another attempt can be made in the site preparation.

6.1.2 Handling Goal Delay Exceptions

Let us now illustrate how the exception handling mechanism allows to capture also the second kind of exception that may occur during the house building, i.e., a delay in the achievement of goal `windows_fitted` w.r.t. to the scheduled time of two weeks. We can easily specify a recovery strategy to deal with the exception, as reported in Listing 6.4.

```
1 <recovery-strategy id="rsWindowDelay">
2     <notification-policy id="npWindowDelay">
3         <condition type="goal-delay">
4             <condition-argument id="target" value="windows_fitted" />
5         </condition>
6         <exception-spec id="windows_delay_exception">
7             <exception-argument id="weeksOfDelay" arity="1" />
8         </exception-spec>
9         <goal id="notify_windows_fitting_delay" />
```

```
10      </notification-policy>
11      <handling-policy id="hpWindowDelay">
12          <condition type="custom">
13              <condition-argument
14                  id="formula"
15                  value="thrown(_,windows_delay_exception,_,Args)
16                         &amp; .member(weekOfDelay(D),Args) &amp; D &gt;= 2"
17               />
18          </condition>
19          <goal id="handle_windows_fitting_delay" />
20      </handling-policy>
21  </recovery-strategy>
```

Listing 6.4: Recovery strategy targeting a delay in windows fitting in the *building-a-house* scenario.

The notification policy inside the recovery strategy specifies an exception spec `windows_delay_exception` with an argument `weeksOfDelay`. By this, we request to the agent throwing the exception to provide an estimation of the expected weeks of delay, in order to calibrate the handling actions accordingly.

Indeed, following the functional decomposition (Figure 6.1), goal `windows_fitted` must be pursued in parallel with two other goals: `roof_built` and `doors_fitted`. While the latter is to be achieved in two weeks, the former takes more time, with a time to fulfill of four weeks. A designer can then define the recovery strategy in a way that the handling policy is applied only if the estimated delay exceeds two weeks. The idea beyond this is that, if the delay amounts to less than two weeks, the subsequent goals in the scheme are not impacted, because they still depend on `roof_built`, which takes four weeks to be achieved. In this particular case, even if the exception occurs, we raise awareness about it, but no corrective action is needed.

We can obtain the behavior described above by specifying a custom condition for the handling policy included in the recovery strategy, as reported at Lines 12-18. The condition is directly expressed as a NOPL formula in the condition argument[9]. As a result, the handling policy is applied (and the catching goal is enabled) only if the

[9]Since in $\mathcal{M}$OISE the organizational specification is encoded in XML, some characters possibly occurring in NOPL formulas (such as &, >, and <) need to be escaped.

number of weeks of delay expressed as argument while throwing the exception is greater than 2.

6.1.3 Exception Handling vs Message Passing

One might argue that robustness could be achieved in agent systems by simply relying on inter-agent messages. Message passing, however, brings on the system some substantial drawbacks. In first lieu, it strengthen agent coupling, as the following example highlights. Listings 6.5 and 6.6 show an implementation of the *site prep contractor* and *engineer* agents, respectively, where the exceptional situation due to `flooding` is handled through message passing.

```
1  +obligation(Ag,_,done(_,site_prepared,Ag),_)
2       : .my_name(Ag)
3      <- !site_prepared;
4         goalAchieved(site_prepared).
5
6  +!site_prepared
7     <- prepareSite.
8
9  -!site_prepared
10      : group(G,house_group,_) &
11        play(Eng,engineer,G) &
12        play(HouseOwner,house_owner,G)
13     <- .send(Eng,tell,
14              exception(site_preparation_exception,[errorCode(flooding)]));
15        .send(HouseOwner,tell,
16              exception(site_preparation_exception,[errorCode(flooding)]));
17        .fail.
18
19 +handled(site_preparation_exception)
20     <- goalAchieved(site_prepared).
```

Listing 6.5: Code of the *site prep contractor* agent, with exception handling realized through message passing.

The raising of an exception may be replaced by the sending of a message to notify the occurrence of a failure, and the corresponding error code. The point is that, since the responsibilities concerning the handling of such a situation are not clearly distributed among the agents, *site prep contractor* might not even know to whom such a message should be sent. Thus, in principle, such a notification should be

broadcasted to all the agents. For the sake of simplicity, however, let us assume that *site prep contractor* knows that *engineer* and *house owner* might be willing to be notified about a failure in the preparation of the site. Thereby, in Listing 6.5, *site prep contractor* sends them the same notification message (Lines 13-16). However, by doing this, we increase the coupling between the involved agents because the recipients of the message, as well as the shape of the message, are hard-coded inside the agent itself.

More critically, the lack of an explicit distribution of responsibilities implies that the the *site prep contractor* cannot have any rightful expectation about the behavior of *engineer* upon reception of its message. In fact, the organization cannot issue any obligation upon *engineer*; the agent might not even be equipped with the right capabilities to handle the exception successfully.

At the same time, agent development cannot follow a uniform approach to address the achievement of organizational goals and the handling of exceptions. Each exception must be addressed by defining *ad hoc* interaction protocols, whose specification falls outside the scope of the organization. To draw an analogy, this message-based solution bears similarities with the definition of functions, in programming languages, where a particular return value denotes a failure (e.g., -1 is the typical failure value of Unix system calls). Of course, the implementation is possible, the drawback is that since the semantics of the values that are returned is twofold, the code will be burdened with checks (i.e., `if` statements) on the return value of every critical function to determine whether the function did its job or not. The same happens for agents, see for instance the code of *engineer* snipped in Listing 6.6.

```
1  +exception(site_preparation_exception,Args)
2       : .member(errorCode(flooding),Args) &
3         group(G,house_group,_) &
4         play(SPC,site_prep_contractor,G)
5      <- performSiteAnalysis(Result);
6         fixFlooding(Result);
7         .send(SPC,tell,handled(site_preparation_exception)).
8
9  +exception(site_preparation_exception,Args)
10      : .member(errorCode(archaeologicalRemains),Args) &
11        group(G,house_group,_) &
12        play(SPC,site_prep_contractor,G)
```

```
13      <- delimitSite;
14         carefullyRemoveRemains;
15         resetGoal(site_prepared).
```

Listing 6.6: Code of the *engineer* agent, with exception handling realized through message passing.

These two plans are specifically devised to capture the message from *site prep contractor*, and are not programmed by following the responsibilities of the agent (i.e., the set of obligations it has to fulfill), but hard coded as "`if`". Our mechanism, instead, allows to program agents just by looking at their responsibilities, capturing in a homogeneous way both the normal and exceptional behavior.

Moreover, since the exception handling is seamlessly integrated within the organization management infrastructure, and the runtime behavior of JaCaMo's normative engine (which issues obligations) has not been substantially changed, exception handling is delivered at no significant additional computational cost w.r.t. "standard" JaCaMo organizations (i.e., without exception handling put in place).

6.2 Leveraging Feedback: Bakery

To further illustrate the flexibility of the proposed approach, let us now consider another scenario. The aim of this example is to highlight the benefits of the availability of contextual feedback, coming from an informed source and in an agreed format, to effectively handle exceptions in a distributed setting, where each agent may have a different and partial visibility over the environment and over the ongoing execution.

In a bakery, among the different products, customers can order cakes which are produced when requested. Cake production involves multiple steps: first the workspace must be set up by gathering the ingredients and switching on the oven, then ingredients must mixed and finally the cake baked.

We can effectively model the bakery as a JaCaMo organization. Agents taking part in the organization will play the following roles: *customer*, *baker*, *supplier*,

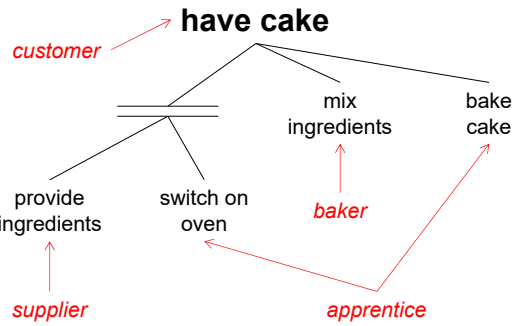

Figure 6.3: Functional decomposition of the *bakery* organizational goal.

and *apprentice*. The cake production can be modeled as a social scheme to be instantiated every time a cake is requested by a customer, as reported in Figure 6.3 and in Listing 6.7.

```
<scheme id="cake_sch">
    <goal id="haveCake">
        <plan operator="sequence">
            <goal id="workspaceSetup">
                <plan operator="parallel">
                    <goal id="provideIngredients" />
                    <goal id="switchOnOven" />
                </plan>
            </goal>
            <goal id="mixIngredients" />
            <goal id="bakeCake" />
        </plan>
    </goal>
    <mission id="mApprentice" min="1" max="1">
        <goal id="switchOnOven" />
        <goal id="bakeCake" />
    </mission>
    <mission id="mSupplier" min="1" max="1">
        <goal id="provideIngredients" />
    </mission>
    <mission id="mBaker" min="1" max="1">
        <goal id="mixIngredients" />
    </mission>
    <mission id="mCustomer" min="1" max="1">
        <goal id="haveCake" />
    </mission>
</scheme>
```

Listing 6.7: Social scheme for producing a cake in the *bakery* scenario.

While the *customer* will be in charge of the top-level goal `haveCake`, the *baker* will have responsibility for the difficult part, namely mixing the ingredients wisely. The

bakery *supplier*, in turn, will be responsible for providing the ingredients, and an *apprentice* for the simpler tasks, i.e., `switchOnOven` and `bakeCake`. The missions featured in the scheme specification encode this responsibility distribution.

We now show how our proposed exception handling mechanism enables a fundamental transfer of contextual information from the context in which an exception occurs to the context in which it must be handled and that such transfer is of primary importance for implementing a successful recovery.

Let us consider, as an illustration, an exceptional situation which is very likely to occur in a real-world scenario, i.e., the unavailability of one or more ingredients. The result would be an impossibility, for the *supplier* agent to achieve its assigned goal `provideIngredients`. The recovery strategy in Listing 6.8 targets this eventuality.

```
<recovery-strategy id="rsIngredients">
    <notification-policy id="npIngredients">
        <condition type="goal-failure">
            <condition-argument id="target" value="provideIngredients" />
        </condition>
        <exception-spec id="ingredientsUnavailable">
            <exception-argument id="missingIngredients" arity="1" />
        </exception-spec>
        <goal id="notifyIngredientsUnavailability" />
    </notification-policy>
    <handling-policy id="handlerIngredients">
        <condition type="always" />
        <goal id="dealWithMissingIngredients" />
    </handling-policy>
</recovery-strategy>
```

Listing 6.8: Recovery strategy targeting the `ingredientsUnavailable` exception in the *bakery* organization.

The notification policy is applied as soon as a goal failure concerning `provide Ingredients` is detected and requires (the *supplier*) to throw an `ingredients Unavailable` exception, providing a list of missing ingredients as exception argument.

Leveraging this information, the *baker* agent can be developed so as to handle the exception, e.g, by using some other ingredient available in the food storage, if any.

For instance, should strawberries be unavailable, we might want the baker use raspberries, if available. The plans below realizes such a behavior.

```
+!dealWithMissingIngredients
     : exceptionThrown(_,ingredientsUnavailable,_) &
       exceptionArgument(_,ingredientsUnavailable,missingIngredients(I)) &
       .member(strawberries,I) &
       available(raspberries)
    <- println("I'll use raspberries instead of strawberries");
       takeFromStorage(raspberries);
       goalReleased(provideIngredients).
```

Listing 6.9: Plans to handle an `ingredientsUnavailable` exception in the *baker* agent.

In other words, thanks to the feedback provided by the supplier (directly involved in the exceptional situation), the baker can leverage the relevant information concerning the exception (in this case the missing ingredient) and combine it with its local knowledge (the food storage provision) in order to put in place the most suitable actions to restore the normal execution of the social scheme.

6.2.1 Support for Collective Exception Handling

We recall that in JaCaMo multiple agents may play the same role. This feature, combined with the exception handling mechanism, allows one to put in place patterns of collective exception handling in which multiple agents may be required to jointly raise or handle exceptions.

For instance, we may want to model that the bakery collaborates with multiple suppliers. We will have multiple agents playing the *supplier* role. Each agent, as a role player, will be then required, when needed, to provide its ingredients. Should, for any reason, goal `provideIngredients` fail, all the *supplier* agents would be required to achieve the throwing goal, thereby each one raising an exception. Each agent will likely provide the list of missing ingredients according to its local knowledge. This feature enables the gathering of feedback from multiple sources, each one having a partial and peculiar perspective over the exception at hand.

The same holds for the exception handling phase. A given catching goal may be assigned to multiple agents playing the same role. In this way, we can model the fact

that a given exception, to be properly handled, requires the execution of the same actions by multiple agents. Consider, for instance, an exception resulting from the failure of goal `switchOnOven` and denoting a fire breakout. All the agents working in the bakery should be asked to leave the room immediately and call the firemen. This eventuality can be well modeled by introducing an additional role, e.g., *worker*, generalizing both the *baker* and the *apprentice*. The following recovery strategy then serves the purpose.

```
<recovery-strategy id="rsOven">
    <notification-policy id="np1">
    <condition type="goal-failure">
        <condition-argument id="target" value="switchOnOven" />
        </condition>
        <exception-spec id="ovenBorken">
            <exception-argument id="status" arity="1" />
        </exception-spec>
        <goal id="notifyIngredientsUnavailability" />
    </notification-policy>
    <handling-policy id="handlerFire">
        <condition type="custom">
            <condition-argument id="formula"
                value="thrown(_,ovenBroken,_,Args) &amp;
                       .member(status(fire),Args)" />
        </condition>
        <goal id="evacuation">
            <plan operator="parallel">
                <goal id="leaveRoomImmediately" />
                <goal id="call911" />
            </plan>
    </handling-policy>
    <handling-policy id="handlerHeat">
        <condition type="custom">
            <condition-argument id="formula"
                value="thrown(_,ovenBroken,_,Args) &amp;
                       .member(status(noHeat),Args)" />
        </condition>
        <goal id="notifyTechSupport" />
    </handling-policy>
</recovery-strategy>
```

Listing 6.10: Recovery strategy targeting the `ovenBroken` exception in the *bakery* organization.

The strategy encompasses two handling policies, to be applied depending on the outcome of the exception raising phase. In particular the first one targets a fire caused by the oven malfunctioning. In this case, we can associate the responsibility

of catching goals `leaveRoomImmediately` and `call911` to the *worker* role. As a result, the corresponding obligations will be issued towards all the role player agents.

By defining multiple handling policies for a given exception, we can also model the fact that the exception should be handled by different agents in different circumstances, as expressed by the following missions.

```
<mission id="mWorker" ... >
    <goal id="leaveRoomImmediately" />
    <goal id="call911" />
</mission>
<mission id="mBaker" ... >
    <goal id="notifyTechSupport" />
</mission>
```

Listing 6.11: Missions for catching goals handling the `ovenBroken` exception in the *bakery* organization.

Through norms, we can assign the first mission to all the workers and the second one to the baker only. The actual handling will involve different agents depending on which feedback is provided as exception argument and, consequently, which policy is applied.

It's worth noting that it is possible to specify complex courses of action to raise and handle exceptions, too, and leverage the organizational infrastructure to coordinate the different collaborating agents. Throwing goals and catching goals, like standard goals, can be complex and encompass multiple sub-goals. The responsibility of such sub-goals can be then distributed among different agents. This enables the use of the normative system to model both the normal and exceptional behavior of the system, uniformly.

We have already shown this mechanism in operation, for catching goals, in Section 6.1.1. Indeed, in that case, exception handling involved the collaboration of an *engineer* agent and of the *house owner* to carry out multiple tasks, concurrently. In Listing 6.10, as well, catching goal `evacuation` encompasses multiple steps.

The same applies to throwing goals included in notification policies. For instance, before throwing an exception, it could be necessary to gather some sensory data,

elaborate it, and finally produce the feedback (i.e., the exception). In case of complex throwing goals, we require that an exception compliant with the corresponding exception spec is actually thrown by the responsible agents before the root of the throwing goal is marked as achieved (see Listing 5.22).

6.2.2 Support for Concerted Exception Handling

Another interesting feature of the mechanism is that it allows to capture the composition of multiple exceptions to be handled jointly, in a similar way to what proposed with the notion of concerted exception handling in SaGE (see Section 3.4). Let us consider again a malfunction in the oven, causing the failure of goal `switchOnOven`. While the two exceptions, related to the ingredients and the oven, may be handled in isolation by the bakery staff, the failure of both `provideIngredients` and `switchOnOven` may cause a concerted `cakePreparationException`, involving the customer, too, in its handling. The recovery strategy in Listing 6.12 models this possibility.

```
<recovery-strategy id="rsConcerted">
    <notification-policy id="...">
        <condition type="custom">
            <condition-argument id="formula"
                     value="scheme_id(S) &amp;
                            failed(S,provideIngredients) &amp;
                            failed(S,switchOnOven)" />
        </condition>
        <exception-spec id="cakePreparationException">
            <exception-argument id="discountCode" arity="1" />
        </exception-spec>
        <goal id="notifyCakePreparationException" />
    </notification-policy>
    <handling-policy id="...">
        <condition type="always" />
        <goal id="cancelOrder" />
    </handling-policy>
</recovery-strategy>
```

Listing 6.12: Recovery strategy targeting the `cakePreparationException` in the *bakery* organization.

The notification policy is triggered only when both goals `provideIngredients` and `switchOnOven` are marked as failed. We can update the scheme missions in order to

assign the responsibility for the throwing goal `notifyCakePreparationException` to the *baker* agent and for the catching goal `cancelOrder` to the *customer*.

As a result, when the throwing goal is enabled, the *baker* is required to inform the *customer* about the fact that cake preparation could not be completed. As a compensation, a discount code for a subsequent purchase, which constitutes the feedback, must be emitted (see Line 10). The concerted exception is finally handled by the *customer* by canceling the order. Concretely, this may amount, e.g., to releasing the root goal `haveCake` or even destroying the scheme instance.

Concerted exception handling is particularly useful to handle the failure of multiple goals inside a choice. Consider, as an illustration, the following goal.

```
<goal id="completePayment">
    <plan operator="choice">
        <goal id="payWithCard" />
        <goal id="payWithCash" />
    </plan>
</goal>
```

Listing 6.13: Complex goal involving a choice.

A payment could be completed either by using a credit card or by cash. In this case, one might want to put in place a recovery strategy to cancel the corresponding order only if both payment attempts fail, i.e., concerting the handling of exceptions coming from the failures of `payWithCard` and `payWithCash`.

The following recovery strategy captures well this eventuality.

```
<recovery-strategy id="rsPayment">
    <notification-policy id="...">
        <condition type="custom">
            <condition-argument id="formula" value="scheme_id(S) &amp;
                                                    failed(S,payWIthCard) &amp;
                                                    failed(S,payWithCash)" />
        </condition>
        ...
    </notification-policy>
    <handling-policy id="...">
        <condition type="..." />
        <goal id="cancelOrder" />
    </handling-policy>
</recovery-strategy>
```

Listing 6.14: Recovery strategy for concerted exception.

6.3 Comparing Exception Handling in JaCaMo and BPMN

In Section 2.4 we introduced Business Processes. BPs realize a business goal by coordinating the tasks undertaken by multiple interacting parties. Given such a distributed nature, multi-agent organizations are a promising paradigm for conceptualizing and implementing them. In order to provide the right support to BPs, however, MAOs must be equipped with a systematic way to model error events denoting the impossibility to complete an activity, which, as explained, realize the exception handling mechanism put in place by BPMN. In this section we show, by illustrating two real-world use cases, how our proposed exception handling mechanism allows to effectively realize business processes encompassing exceptions in JaCaMo.

Recalling Weske's definition (Weske, 2007), a business process is "a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal." In general, such business goal is achieved by breaking it up into sub-goals, which are distributed to a number of actors. Each actor carries out part of the process, and depends on the collaboration of others to perform its task. One limit of business processes is that they integrate, at the same abstraction level, both the business logic and the interaction logic (message passing). Multi-agent systems, and in particular models for multi-agent organizations, are promising candidates to supply the right abstractions to keep processes linked together in a way that allows modeling the overall system in terms of goals rather than of messages.

The potential of multi-agent systems and organization for the realization of business processes has been advocated in (Sabatucci and Cossentino, 2019; Cossentino et al., 2020; Cossentino et al., 2021), as well. The authors propose a tool for the automatic generation of $\mathcal{M}$OISE organizations from the BPMN specification of business processes, in the context of dynamic workflows. The approach is complementary to ours and can pave the way to a fruitful concrete application of an agent-oriented approach to support the functioning of modern enterprises.

6.3.1 Translating BPMN Processes into JaCaMo Organizations

When implementing a business process as a JaCaMo organization, a designer has to be aware of a substantial difference between the two underlying paradigms. A business process describes an activity flow where choices, upon alternative execution branches, depend on the outcome of the activities performed that far. Instead, in JaCaMo each organization generally has a complex goal, whose structure is provided as a functional decomposition into sub-goals. The functional decomposition is used to track and guide the execution, understanding when a sub-goal is to be pursued and emitting the corresponding obligation.

The implementation of business processes through JaCaMo organizations, thus, requires some special treatment, especially for what concerns the BPMN gateways, where exclusive choices upon data are taken. Specifically, we capture these gateways and their alternative branches as *special goals* within the functional decomposition. Considering a choice, the goals amounting to the various alternatives are mutually exclusive: the achievement of one of such alternative goals determines a specific execution path that constrains the evolution of the remainder of the social scheme. This stratagem allows incorporating, at least in part, within a functional decomposition the execution flow based on data. A dedicated *manager* agent will be in charge of satisfying the obligations issued upon such special goals.

Having this in mind, the following steps provide a guideline to map a number of interacting business processes into a JaCaMo organization.

1. For each process, a corresponding *manager* role in the organization is defined. The agent(s) playing this role will have to decide on the alternative branches to choose in the process execution;
2. For all the activities in a process, suitable *worker* roles in the organization are defined. These roles will be played by the agents in charge of executing the activities;
3. For each process:

- A group collecting the manager and all the workers involved in the process is defined;
- A social scheme is created to organize the activities as a goal decomposition tree[10]. Corresponding missions are defined, to be assigned to roles of the group in charge of the process by defining suitable norms;

4. For each set of activities to be executed in sequence, the corresponding goals are added to the social scheme by means of the "sequence" operator;
5. For each set of activities to be executed without strict ordering, the corresponding goals are added to the social scheme by means of the "parallel" operator;
6. If a choice is present inside a process (either an 'or' or 'exclusive or' gateway), a corresponding goal is added to the social scheme by means of the "choice" operator. Each sub-goal represents a possible course of action (alternative branch). Every alternative in the choice should include a special goal, encoding the chosen path to be assigned to the process manager. Depending on which goal will be achieved by the manager, the execution will follow a branch or another;
7. If a process sends a message that makes another process start, the message should be sent to the process manager, which, as a consequence, will instantiate the social scheme corresponding to the process.

6.3.2 Error Events as Recovery Strategies: Incident Management

The *incident management* case (Object Management Group, 2021), in Figure 6.4, that we use as a first running example, models the interaction between a customer and a company for the management of a problem that was reported by the customer. It involves several interacting processes. The Customer reports the problem to a Key Account Manager who, based on its experience, can either solve the problem

[10]Here, we restrict our attention to processes that do not include loops; otherwise it would not be possible to express them as decomposition trees.

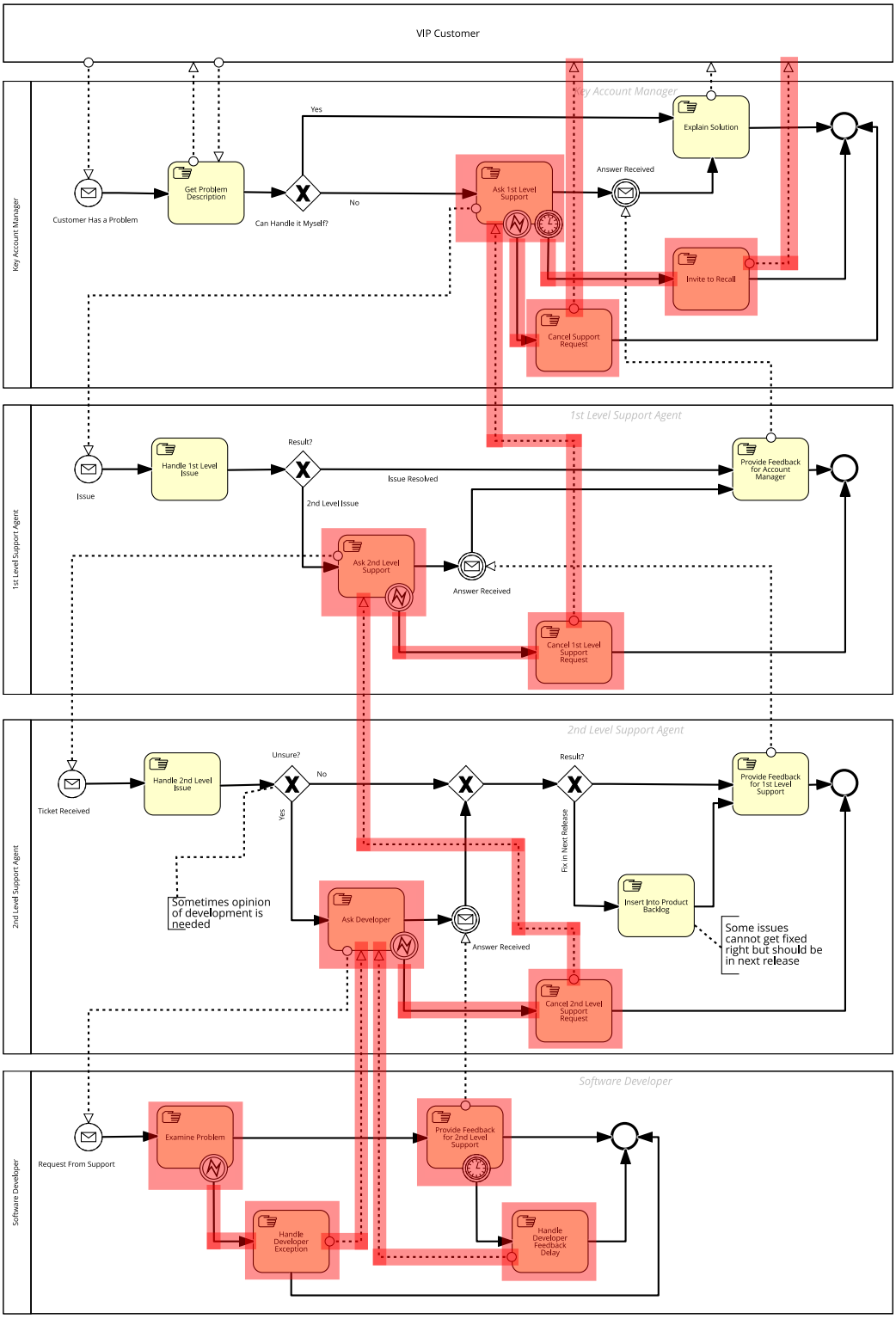

Figure 6.4: The *incident management* BPMN diagram enriched with exception management. The exceptional flow is highlighted in red.

directly or ask for the intervention of a First Level Support process. The problem can, then, be recursively treated by different support levels until, in the worst case, it is reported to the Software Developer. Here, the business aim of the process, i.e., to solve the reported problem, is decomposed and distributed over up to five BPMN processes, whose execution requires interaction and coordination – realized in this case through message exchange.

As Figure 6.4 highlights (in red), the case includes treatment of exceptional situations in all of the involved processes. For instance, the `Examine Problem` activity in the Software Developer process might trigger an error event, causing the `Handle Developer Exception` activity to be executed. As a result, the exception is then propagated upwards through message passing, possibly causing the occurrence (and handling) of additional exceptions. The reception of the failure message sent by the Developer, in fact, will likely cause the `Ask Developer` activity in the Second Level Support process to fail, as well, triggering a further exception. Exceptions are propagated in a uniform way upwards until the customer is notified about the impossibility to solve the problem and the support request is canceled.

The scenario also includes a set of timeout events, which, as we will see, can be well modeled as deadline expiration exceptions with our exception handling approach. For instance, activity `Provide Feedback for 2nd Level Support` encompasses a timeout event triggering a dedicated activity to handle the delay.

Let us now explain how the *incident management* scenario can be mapped into a JaCaMo organization by applying the steps above. For the sake of simplicity, let us consider the Key Account Manager process. For the other processes we follow a similar approach. First, we introduce a manager role *am* and, for each activity in the process, we define a corresponding worker. aw_1, for instance, will be in charge of `Get Problem Description`, aw_2 will be in charge of `Explain Solution`, and so on. As a further step we define a Key Account group collecting *am* and all of its workers. At the beginning of the execution, the customer agent will send a message to *am* reporting the problem. As a consequence, the agent playing role *am* will instantiate a scheme that will be assigned to this group and will encompass the overall Key

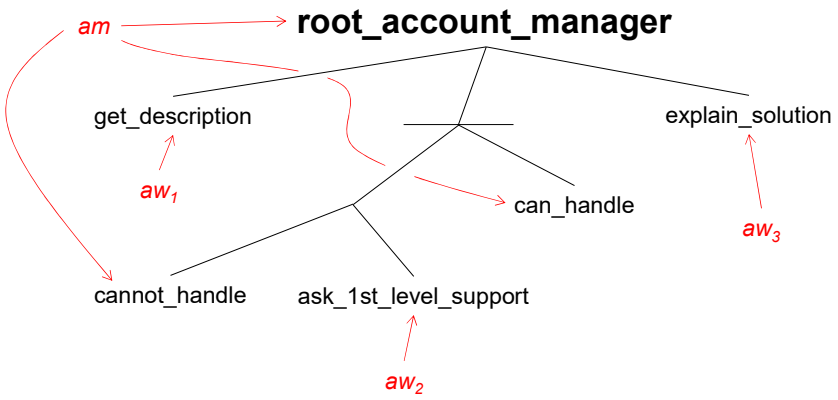

Figure 6.5: Social scheme realizing the Key Account Manager process.

Account Manager process. The root goal of such scheme will be assigned to *am* that, in order to satisfy it, will have to manage the successful execution of the social scheme. *am* will be in charge of the goals introduced to determine which branch of a given gateway to follow, as well (in this case `can_handle` and `cannot_handle`).

Figure 6.5 shows the scheme available for instantiation to *am* that represents the possible courses of actions during the execution of the Key Account Manager process. Goals including a choice are grouped under a single horizontal line. The Figure also shows which agents such goals are assigned to (through missions and norms).

The scheme is to be instantiated as soon as a problem to be solved is reported. As a consequence, the obligation to achieve goal `get_description` will be issued to the corresponding worker aw_1. After the successful achievement of `get_description`, two obligations under a choice will be issued towards *am*: the former related to the `can_handle` goal and the latter to `cannot_handle`. Depending on the result of the previous activity (i.e., whether the problem requires further support or can be handled at that level), *am* will decide to achieve either one of the two goals, the choice made by *am* thus constrains the subsequent obligations that will be generated. In the former case, the normative system will simply issue the obligation to `explain_solution`, while in the latter it will issue the obligation to ask support to the first level.

In the second case, according to the BPMN diagram, again two options are available. If an answer is received from the first level support, the solution has to be explained, as well. On the contrary, if a feedback is not received after one day, causing a timeout, an invitation to recall should be sent to the customer. Similarly, the customer request must be canceled if an exception is propagated from the following support level. In the following, we will see how both eventualities can be captured through exception handling.

As said before, the request for support from the first level is concretely realized by the responsible worker by instantiating the scheme corresponding to the First Level Support process. The very same agent, in this second scheme, will play the manager role. The scheme will then progress in a similar way as for the Key Account Manager.

Let us suppose that the problem at hand requires the propagation of the support request down until the Developer process. Listing 6.15 shows an excerpt of the scheme specification realizing such process.

```
<scheme id="scheme_developer">
    <goal id="root_developer">
        <plan operator="sequence">
            <goal id="examine_problem" />
            <goal id="provide_feedback_for_2nd_level_support" ttf="1 day"/>
        </plan>
    </goal>
    <recovery-strategy id="rsDeveloper1">
        <notification-policy id="npDeveloper1">
            <condition type="goal-failure">
                <condition-argument id="target" value="examine_problem" />
            </condition>
            <exception-spec id="developer_exception">
                <exception-argument id="warrantyStatus" arity="1" />
            </exception-spec>
            <goal id="raise_developer_exception" />
        </notification-policy>
        <handling-policy id="hpDeveloper1">
            <condition type="always" />
            <goal id="handle_developer_exception" />
        </handling-policy>
    </recovery-strategy>
    <recovery-strategy id="rsDeveloper2">
        <notification-policy id="npDeveloper2">
            <condition type="goal-ttf-expiration">
```

```
                <condition-argument id="target"
                      value="provide_feedback_for_2nd_level_support" />
            </condition>
            <exception-spec id="developer_feedback_delay">
                <exception-argument id="reason" arity="1" />
            </exception-spec>
            <goal id="raise_developer_feedback_delay" />
        </notification-policy>
        <handling-policy id="hpDeveloper2">
            <condition type="always" />
            <goal id="handle_developer_feedback_delay" />
        </handling-policy>
    </recovery-strategy>
    ...
</scheme>
```

Listing 6.15: Social scheme realizing the Software Developer process.

The scheme foresees two recovery strategies, allowing us to effectively model both the error event possibly occurring in activity `Examine Problem` and the timeout event in activity `Provide Feedback for 2nd Level Support`. The first strategy, indeed, is to be activated when a failure occurs involving goal `examine_problem`, which reifies the same name activity in BPMN and its associated error event. The second one, in turn, is applied when a delay (bringing to a deadline expiration) occurs in the achievement of goal `provide_feedback_for_2nd_level_support`, modeling the timeout of the corresponding activity.

Listings 6.16 and 6.17 show the code of the two worker agents, that are responsible, respectively for goals `examine_problem` and `provide_feedback_for_2nd_level_support`, and for possibly throwing the related exceptions.

```
+obligation(Ag,_,done(_,examine_problem,Ag),_)
     : .my_name(Ag)
    <- !examine_problem; // set speed to 70%
       goalAchieved(examine_problem).

+!examine_problem
    <- // check warranty, problem description, manual, ...

-!examine_problem
    <- goalFailed(examine_problem);
       .fail.

+obligation(Ag,_,done(_,raise_developer_exception,Ag),_)
     : warrantyStatus(S)
```

```
15      <- throwException(developer_exception,[warrantyStatus(S)]);
16         goalAchieved(raise_developer_exception).
```

Listing 6.16: Code of the first worker in the *incident management* scenario.

```
1  +obligation(Ag,_,done(_,provide_feedback_for_2nd_level_support,Ag),_)
2       : .my_name(Ag)
3      <- ...
4
5  +obligation(Ag,_,done(_,raise_developer_feedback_delay,Ag),_)
6      <- throwException(developer_feedback_delay,[reason(...)]);
7         goalAchieved(raise_developer_feedback_delay).
```

Listing 6.17: Code of the second worker in the *incident management* scenario.

Both exceptions, amounting to the error and timeout events, are modeled uniformly, by issuing an obligation to achieve the corresponding throwing goal towards the responsible agent (see the last plans in the agents' code). The only difference is that for what concerns the error event, the exception is modeled as a goal failure, thereby requiring an explicit execution of the `goalFailed` operation by the first worker (see Line 10 in Listing 6.16). In the second case, conversely, the exception is modeled as a goal delay, not requiring any explicit action by the agent. The obligation concerning the throwing goal is issued as soon as the deadline of goal `provide_feedback_for_2nd_level_support` expires. The two exception specifications encompasses two exception arguments, the warranty status in the former case, and a reason for the delay in the latter.

Listing 6.18, finally shows a piece of code of the agent playing the manager role in the scheme. We recall that the very same agent is the one that is responsible for `ask_developer` in the Second Level Support process scheme. The instantiation of the Developer scheme by the agent concretely realizes the support request, as illustrated before with the Key Account Manager. In other words, the worker in charge of `ask_developer` leverages the Developer scheme (and its coordinated execution) as a tool to successfully complete its own activity.

```
1  +obligation(Ag,_,done(_,ask_developer,Ag),_)
2       : .my_name(Ag) & ...
3      <- createScheme(_, scheme_developer,_);
4         ...
5
```

```
6  +feedback(F)
7      <- goalAchieved(ask_developer).
8
9  +obligation(Ag,_,done(_,handle_developer_exception,Ag),_)
10      : .my_name(Ag)
11     <- goalFailed(ask_developer);
12        goalReleased(root_developer);
13        goalAchieved(handle_developer_exception).
14
15 +obligation(Ag,_,done(_,handle_developer_feedback_delay,Ag),_)
16      : .my_name(Ag)
17     <- goalFailed(ask_developer);
18        goalReleased(root_developer);
19        goalAchieved(handle_developer_feedback_delay).
20
21 +obligation(Ag,_,done(_,raise_2nd_level_support_exception,Ag),_)
22      : .my_name(Ag)
23     <- ...
24
25 ...
```

Listing 6.18: Code of the developer manager agent in the *incident management* scenario.

The first plan realizes activity `Ask Developer` and encompasses the scheme instantiation (see Line 3). The goal is marked as achieved as soon as a feedback is received from the developer layer, as a result of the second plan.

The third and fourth plans, at Lines 9 and 15 realize the handling of the two exceptions which could be thrown in the Developer process. Here, both exceptions are treated in the same way, causing the failure of activity `Ask Developer`. For this reason the corresponding goal is marked as failed. In this way, the exception is propagated from the Developer process to the Second Level Support, because the failure will likely activate an additional recovery strategy in such scheme, recursively requiring the throwing of a further exception towards the first level of support. The last plan, at Line 21 realizes this behavior.

In short, should an exception occur at any level of support, it will be in the end propagated upwards until it will reach the agent managing the Key Account Manager scheme, which will finally cancel the support request from the customer of invite it to recall, according to the recovery strategies defined for the given scheme.

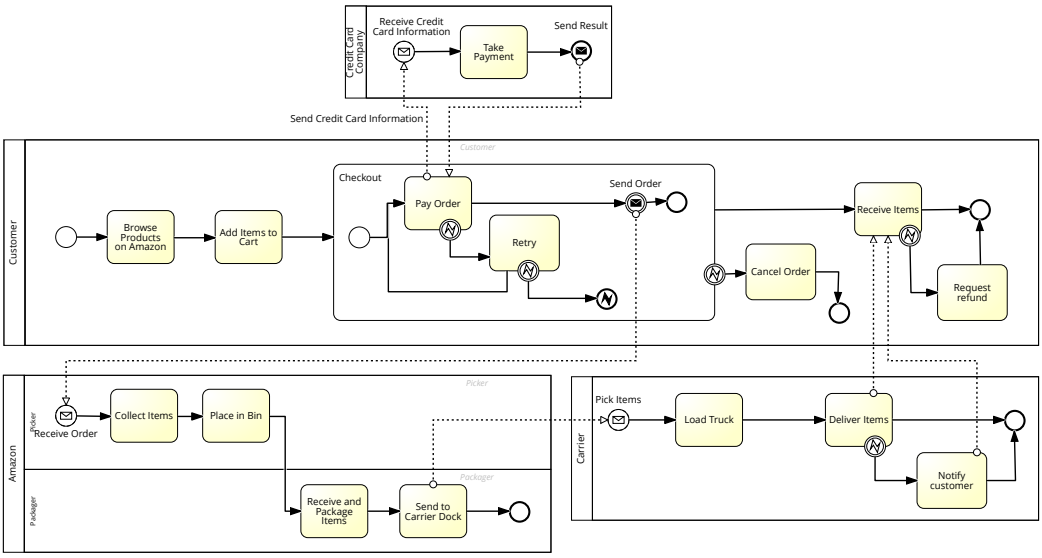

Figure 6.6: The *Amazon order fulfillment* BPMN diagram.

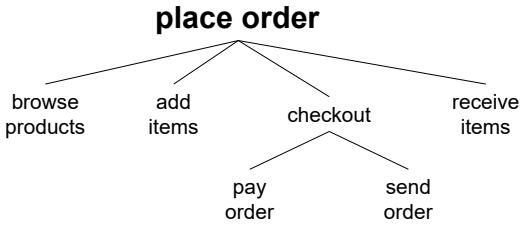

Figure 6.7: The Customer process organizational scheme in the *Amazon order fulfillment* scenario.

6.3.3 Modeling Recurrent Exception Handling: Order Fulfillment

Another well-known BPMN use case, inspired from the real world, is the *Amazon order fulfillment* collaboration (Object Management Group, 2021), Figure 6.6. The business goal of the process is to complete an online purchase and it is decomposed into four interacting processes, each one modeling a specific aspect of the purchase.

As in the previous case, multi-organizations can provide a good computational support for its realization. The case can be effectively modeled as a a JaCaMo organization, in a way similar to what done for the *incident management*. Each process can be modeled as a social scheme in which agents carry out goals amounting to process activities. Also in this case, some processes encompass exceptions, which

have to be addressed. Figure 6.7 shows the social scheme realizing the Customer process, while Listing 6.19 shows the code of the recovery strategies targeting the three exceptions possibly occurring in the process.

```
<recovery-strategy id="rsPay">
    <notification-policy id="npPay">
        <condition type="goal-failure">
            <condition-argument id="target" value="payOrder" />
        </condition>
        <exception-spec id="paymentRefused">
            <exception-argument id="balance" arity="1" />
        </exception-spec>
        <goal id="raisePaymentRefused" />
    </notification-policy>
    <handling-policy id="hpPay">
        <condition type="always"/>
        <goal id="retry" />
    </handling-policy>
</recovery-strategy>
<recovery-strategy id="rsRetry">
    <notification-policy id="npRetry">
        <condition type="goal-failure">
            <condition-argument id="target" value="retry" />
        </condition>
        <exception-spec id="checkoutFailed" />
        <goal id="raiseCheckoutFailed" />
    </notification-policy>
    <handling-policy id="hpRetry">
        <condition type="always"/>
        <goal id="cancelOrder" />
    </handling-policy>
</recovery-strategy>
<recovery-strategy id="rsReceive">
    <notification-policy id="npReceive">
        <condition type="goal-failure">
            <condition-argument id="target" value="receiveItems" />
        </condition>
        <exception-spec id="itemsNotReceived" />
        <goal id="raiseItemsNotReceived" />
    </notification-policy>
    <handling-policy id="hpReceive">
        <condition type="always"/>
        <goal id="requestRefund" />
    </handling-policy>
</recovery-strategy>
```

Listing 6.19: Recovery strategies for the Customer process in the *Amazon order fulfillment* scenario.

It's worth noting that our approach allows modeling in a straightforward way also recurrent exceptions, i.e., exceptions which are thrown during the handling of a previously thrown exception. Consider, for instance, activity `Pay Order`; an exception could occur during its execution, triggering activity `Retry`. This second activity can trigger an exception itself, causing the failure of the expanded subprocess Checkout, and triggering a further activity for handling it, `Cancel Order`.

This pattern is well captured by the first two recovery strategies in Listing 6.19. The first one, in particular addresses the failure of the organizational goal `payOrder` which reifies activity `Pay Order`, and encompasses a catching goal `retry`. Such goal, as any other, may fail, causing the occurrence of an additional exception, that is addressed by the second recovery strategy. Indeed, here the condition triggering the notification policy is exactly a failure of goal `retry` and the handling is realized through goal `cancelOrder`.

This approach has analogies with the propagation of an exception along the call stack in programming languages, or with the *escalate* directive in Akka. In programming languages, it's the call stack that defines the scope within which the information concerning an exception must be propagated. The supervision hierarchy in the actor model serves the same purpose. In our approach, in turn, the responsibilities induced by recovery strategies shape the channel through which the relevant information concerning the exception can flow from one agent to another.

6.3.4 Capturing Other Kinds of Events

As a remark, we highlight that our approach can be easily extended to support the modeling, besides error and timeout events, of other kinds of BPMN events, such as *signals*. A signal, similarly to an error, may be thrown during the execution of an activity and trigger, as a result, the execution of further activities (e.g., they may start an additional process). The main difference is that, while an error denotes the failure of an activity, breaking the normal flow in the execution of the process, a

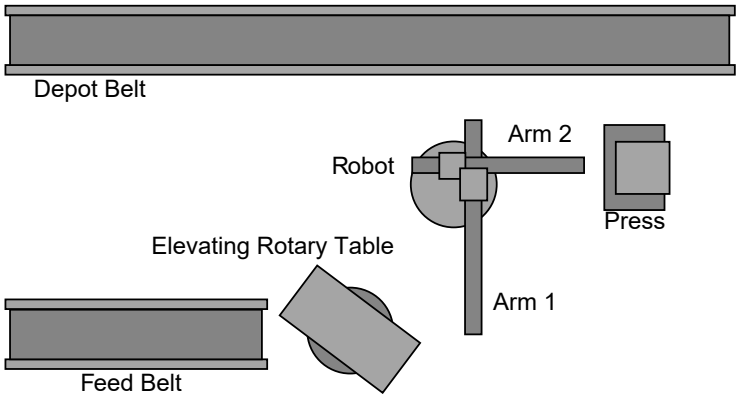

Figure 6.8: Industrial production cell.

signal does not interrupt the normal execution of the process, which continues after the signal is thrown.

To this end, a designer may define custom conditions for triggering notification and handling policies. Such conditions would realize a conceptual mapping between some relevant organizational states and the occurrence of signal events. Recovery strategies encompassing these conditions, would then be applied when such events occur and would specify the additional course of actions to be executed in response to the given signal, without stopping the execution of the original social scheme.

The organizational infrastructure may be extended, as well – e.g., by extending the goal lifecycle or by providing further organizational operations – to capture a wider range of events.

6.4 Exception Handling in an Industrial Scenario: Production Cell

In the context of Industry 4.0 and Smart Factories, one main challenge is to conjugate an increasing level of autonomy of the components involved in the industrial processes with interoperability and flexibility in order to build smart and adaptive infrastructures, able to reconfigure upon the occurrence of perturbations. The aim of this section is to show how exception handling grounded on responsibility can bring concrete benefits in such a context. To this end, we consider an industrial

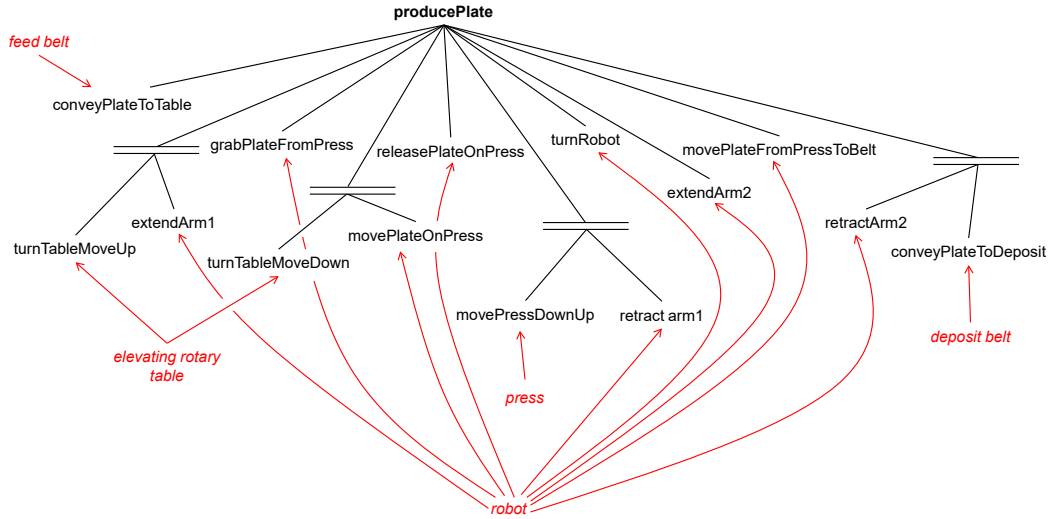

Figure 6.9: JaCaMo scheme for the *production cell* scenario.

scenario, inspired by the well-known and widely used production cell described in (Lewerentz and Lindner, 1995).

A production cell is composed of five machines, see Figure 6.8: two conveyor belts (a feed belt and a depot belt), an elevating rotary table, a press, and a rotary robot equipped with two extensible arms. Each device has a set of sensors that provide information about the environment and a set of actuators. The task of the cell is to get a metal plate from a storage rack via the feed belt, transform it into a forged plate by using the press, and return it to the deposit via the depot belt.

More in detail, the cell should perform the following steps for each metal plate that is provided: 1) the feed belt conveys the plate from the storage rack to the elevating rotary table, 2) the table rotates and lifts to the position where the robot can grab the plate, 3) Arm 1 of the robot extends and picks the plate up, 4) the robot turns and Arm 1 places the plate onto the press, 5) the press forges the plate while the robot turns again, 6) Arm 2 picks up the forged plate and places it on the depot belt, and 7) the depot belt carries the plate forward to the depot.

The described production cell can be well realized as a JaCaMo organization, where each machine is operated by a different agent, and the organization coordinates the production process. Figure 6.9 shows a possible JaCaMo scheme realizing

the production of a metal plate. The figure also shows, in red, which agents are responsible for which organizational goals.

6.4.1 Shortage of resources

The first exception that we consider in this scenario is a delay in the delivery of metal plates. A reduction of the stockpile can be handled by slowing down the production. The idea is that having the production cell come to a complete stop is expensive and to be avoided when possible. To this aim, we can extend the scheme specification with the following recovery strategy.

```
<recovery-strategy id="strStock">
    <notification-policy id="npStock">
        <condition type="goal-ttf-expiration">
            <condition-argument id="target" value="conveyPlateToTable" />
        </condition>
        <exception-spec id="exStock">
            <exception-argument id="availablePlates" arity="1" />
        </exception-spec>
        <goal id="notifyRemainingStock" />
    </notification-policy>
    <handling-policy id="hpStock">
        <condition type="custom">
            <condition-argument id="formula"
                                value="thrown(_,exStock,_,Args) &amp;
                                       .member(availablePlates(N),Args)
                                       &amp; N &lt;= 10" />
        </condition>
        <goal id="slowDownProduction" />
    </handling-policy>
</recovery-strategy>
```

Listing 6.20: Recovery strategy targeting a shortage of resources in the *production cell* scenario.

The strategy models the need to throw an exception as soon as a delay is detected in the achievement of goal `conveyPlateToTable`. In this case, the information to be provided by the agent throwing the exception amounts to the number of remaining plates. To this end, the notification policy encompasses the goal `notifyRemainingStock` as throwing goal. The handling policy, encompassing `slowDownProduction` as catching goal, is to be applied when the number of remaining plates is less than a certain amount.

Let us assume the *robot* is in charge of supervising the production cell, and participates in the exception handling by taking the responsibility for `requestRemainingStock`. We assume that the responsibility for `notifyRemainingStock` is taken by the *feed belt*, since it has access to the local information, concerning the stockpile.

The following listing shows an excerpt of the *feed belt*'s code, concerning throwing the exception.

```
+obligation(Ag,_,done(_,conveyPlateToTable,Ag),_)
     : .my_name(Ag)
    <- pickPlateFromStorage; // action over the environment
       startBelt(10); // start the belt for 10 seconds
       stopBelt;
       goalAchieved(conveyPlateToTable).

+obligation(Ag,_,done(_,notifyRemainingStock,Ag),_)
     : .my_name(Ag) &
       inventory(I) & .member(plates(N),I)
    <- throwException(exStock,[availablePlates(N)]);
       goalAchieved(notifyRemainingStock).
```

Listing 6.21: Implementation of the *feed belt* agent in the *production cell* scenario.

While the first plan allows the agent to fulfill its task of moving the plate on the belt from the storage to the table, the second plan realizes the behavior allowing the agent to fulfill its responsibility to throw the `exStock` exception.

The next listing, in turn, is an excerpt of a possible implementation of the *robot*.

```
+obligation(Ag,_,done(_,slowDownProduction,Ag),_)
     : .my_name(Ag) &
       exceptionThrown(_,exStock,_) &
       exceptionArgument(_,exStock,availablePlates(N)) & N >= 5
    <- setProductionSpeed(0.7); // set speed to 70%
       goalAchieved(slowDownProduction).

+obligation(Ag,_,done(_,slowDownProduction,Ag),_)
     : .my_name(Ag) &
       exceptionThrown(_,exStock,_) &
       exceptionArgument(_,exStock,availablePlates(N)) & N >= 2
    <- setProductionSpeed(0.3);
       goalAchieved(slowDownProduction).

+obligation(Ag,_,done(_,slowDownProduction,Ag),_)
     : .my_name(Ag) &
       exceptionThrown(_,exStock,_) &
       exceptionArgument(_,exStock,availablePlates(N)) & N <= 1
```

```
19      <- stopProduction;
20         goalAchieved(slowDownProduction).
```

Listing 6.22: Implementation of the *robot* agent in the *production cell* scenario.

As soon as the needed information concerning the stockpile is available, the *robot* will be in condition to handle the situation by pursueing goal `slowDownProduction`. The *robot* will, then, have the behavior of the production cell modulated according to amount of plates that are left.

6.4.2 Motor Break

As a second reasonable source of exceptions, let us consider the elevating rotary table: it is moved by two motors, one elevating and one rotating it. Should a malfunction occur, causing the failure of goal `turnTableMoveUp`, it would be desirable for the robot, depending on which motor stopped, to notify the personnel and, thus, quicken the restoration. At the same time, the production cell should be paused. The following recovery strategy effectively serves the purpose.

```
1  <recovery-strategy id="strMotor">
2      <notification-policy id="npMotor">
3          <condition type="goal-failure">
4              <condition-argument id="target" value="turnTableMoveUp" />
5          </condition>
6          <exception-spec id="exMotor">
7              <exception-argument id="motorNumber" arity="1" />
8          </exception-spec>
9          <goal id="notifyStoppedMotorNumber" />
10     </notification-policy>
11     <handling-policy id="hpMotor">
12         <condition type="always" />
13         <goal id="motorFix">
14             <plan operator="parallel">
15                 <goal id="scheduleTableMotorFix" />
16                 <goal id="pauseProduction" />
17             </plan>
18         </goal>
19     </handling-policy>
20 </recovery-strategy>
```

Listing 6.23: Recovery strategy targeting a motor break in the *production cell* scenario.

By distributing the responsibilities for `scheduleTableMotorFix` to the *robot* agent, and for `notifyStoppedMotorNumber` to the *table*, agents are put in condition to effectively cope with the failure.

6.4.3 Risk for Human Being

We conclude the illustration of the *production cell* scenario by showing how exception handling can prove useful to support the execution of sensitive tasks, like the treatment of perturbations arising from the interaction of the machines with a human operator, to preserve safety. In particular, let us now capture the need to change pace when a human operator is detected in the operational area of the press.

```
<recovery-strategy id="strPress">
    <notification-policy id="npHuman">
        <condition type="goal-ttf-expiration">
            <condition-argument id="target" value="movePressDownUp" />
        </condition>
        <exception-spec id="exHuman">
            <exception-argument id="slowdownCode" arity="1" />
            <exception-argument id="humanCoords" arity="2" />
        </exception-spec>
        <goal id="explainSlowdownReason" />
    </notification-policy>
    <handling-policy id="hpHuman">
        <condition type="custom">
            <condition-argument id="formula"
                    value="thrown(_,exHuman,_,Args)
                           &amp; .member(humanCoords(X,Y),Args)
                           &amp; X &lt; 2 &amp; Y &lt; 3" />
        </condition>
        <goal id="emergencyStop" />
    </handling-policy>
</recovery-strategy>
```

Listing 6.24: Recovery strategy targeting the presence of a human operator in the *production cell* scenario.

We suppose that the press is equipped with sensors to detect a human operator working at it and, for the sake of safety, it immediately slows down. Such a slowdown may result in the impossibility to achieve `movePressDownUp` before the deadline. The above listing shows the recovery strategy targeting this eventuality.

The *press* agent will be then required to explain the reasons for the slowdown. In presence of a human operator, the exact position will be provided as well, in order to have the whole production cell calibrate its functioning accordingly, in some cases arriving at a full stop to avoid occasional wounds.

Listing 6.25 shows a plan to be delivered to the agent handling the exception if a human is detected in a dangerous area.

```
1 +obligation(Ag,_,done(_,emergencyStop,Ag),_)
2      : .my_name(Ag) &
3        exceptionArgument(_,exHuman,humanCoords(X,Y))
4     <- pressEmergencyStop;
5        activateAlarm;
6        delimitArea(X,Y);
7        goalAchieved(emergencyStop).
```

Listing 6.25: Agent plan to handle the presence of a human operator in the *production cell* scenario.

6.5 Adapting to Adverse Conditions: Parcel Delivery

As a final illustrating scenario, let us consider a MAS organization supporting the activity of a transportation company. In this setting, the delivery of a parcel involves multiple steps carried out by multiple actors: first, the parcel must be prepared, taking the goods from the warehouse, packing them up, and loading the truck. Then, the parcel can be delivered by (i) locating the address, (ii) planning the route, (iii) reaching the destination, and (iv) handing the parcel over to the customer. At last, the order can be closed. Figure 6.10 shows a JaCaMo scheme for an organization supporting the process above. Suppose that, due to some roads unexpectedly closed, the truck delays the achievement of goal `reachDestination`. It would be desirable that the planner, who is in charge of planning the routes, were notified about the reasons, in order to plan an alternative path for the upcoming delivery. The following strategy captures this need. Line 3 specifies that the exception should be thrown in case of delay in the achievement of the goal `reachDestination`, `reportDelayReason` is the throwing goal, and Line 8 specifies the expected kind of exception. To avoid the occurrence of the same problem in future deliveries, it

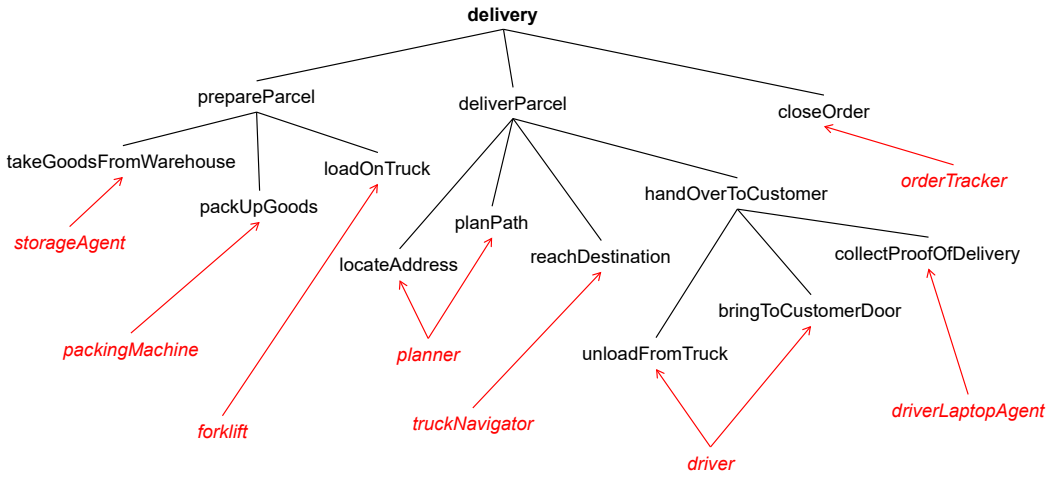

Figure 6.10: JaCaMo scheme for the *parcel delivery* scenario.

would be useful to make the information concerning the closed roads reach the *planner* agent in charge of planning the path. In this way the agent could take it into account in future planning activities, resulting in a performance improvement for the whole organization.

```
<recovery-strategy id="strParcel">
    <notification-policy id="npParcel">
        <condition type="goal-ttf-expiration">
            <condition-argument id="target" value="reachDestination" />
        </condition>
        <exception-spec id="exParcel">
            <exception-argument id="reason" arity="..." />
            <exception-argument id="closedRoads" arity="1" />
        </exception-spec>
        <goal id="reportDelayReason" />
    </notification-policy>
    <handling-policy id="hpParcel">
        <condition type="always" />
        <goal id="updateMap" />
    </handling-policy>
</recovery-strategy>
```

Listing 6.26: Recovery strategy for the *parcel delivery* scenario.

If, as assumed, the reason is that some road is closed, the *planner* will use this information to modify the routes of the agents that perform the deliveries, making the organization adapt to the adverse contextual conditions, as with the following plan.

```
+obligation(_,_,done(_,updateMap,_),_)
     : exceptionArgument(_,exParcel,closedRoads(R))
    <- .addToIgnoreList(R).
```

6.6 Summary and Comparison with Previous Approaches

To conclude the chapter, we summarize the main features of the presented exception handling approach and compare it with the works previously reviewed in Chapters 2 and 3.

Our proposal mainly takes inspiration from exception handling mechanisms proposed in the fields of programming languages and actors research and tries to accommodate such a view with the peculiarities of an agent-oriented approach. More in detail, we adopted the perspective originally put forward by Goodenough, who sees in exception handling a powerful tool to achieve software robustness, promoting at the same time modularity and decoupling.

In this sense, the notions of responsibility and feedback are of uttermost importance. The proposed exception handling mechanism is grounded on both of them. Indeed, recovery strategies shape the scope of the responsibilities that agents taking part in an organization must take on, to enable the relevant feedback concerning perturbations be produced from informed sources and handled by competent ones. At the same time, the approach is seamlessy integrated, both at a conceptual and a programming level, within the organizational framing.

Differently than the approach by Platon et al. (Platon, 2007; Platon et al., 2007a; Platon et al., 2007b; Platon et al., 2008), where exception handling is seen as a tool that the individual agent can activate internally, to preserve self-control despite the occurrence of perturbations, the proposal we have made leverages the distributed nature of exception handling, typical of programming languages and of the actor model, and suited to distributed systems made of cooperative parties, like MAOs.

Contingency plans, in turn, realize Platon's vision in the context of the Jason programming language. They incorporate exception handling in the reasoning cycle of

the agents, allowing them to deal with local failures that cause the impossibility to achieve some internal goals. As highlighted in the practical use cases, the approach is complementary to ours. Our exception handling mechanism, instead, aims at providing support for exceptions that cannot be handled by the agents in isolation. In this sense, the execution of a contingency plan at the agent level may later trigger exception handling at the organizational level.

In (Klein and Dellarocas, 1999; Dellarocas and Klein, 2000; Klein and Dellarocas, 2000), the proposed exception handling service provides sentinels, that are equipped with handlers, to be plugged into existing agent systems. The service actively looks for exceptions in the system and prescribes specific interventions from a body of general procedures. Sentinels communicate with agents using a predefined language for querying about exceptions and for describing exception resolution actions. Agents, for their part, are required to implement a minimal set of interfaces to report on their own behavior and modify their actions, according to the prescriptions given by the sentinels. As a difference, our proposal seamlessly integrates exception handling into the agents themselves, without centralizing it into dedicated sentinels. In this way, it accommodates Goodenough's recommendation that appropriate "fixup" will vary from one use of the operation to the next.

SaGE (Souchon et al., 2003; Souchon, 2005) integrates exception handling in the execution model of the agents. Handlers can be defined and associated with the services provided by an agent, as part of its behavior. However, agents since no responsibilities to be taken by the agents are explicitly encompassed, agents are assumed to be benevolent and cooperative. As a consequence, the framework is not well suited for open systems where agents are self-interested.

Mallya and Singh (Mallya and Singh, 2005b; Mallya, 2005) proposed to model exceptions via commitment-based protocols. Anticipated exceptions, occurring during the execution of an interaction protocol (i.e., deviations from the normal flow that occur often enough and are part of the model), are dealt with by specifying a hierarchy of preferred runs. Preferences can, then, be used to define exceptional runs. Exception handlers are treated as runs just like protocols. Handlers can be

spliced inside a given protocol when an exceptional run is detected. The paper proposes also an approach to deal with unexpected exceptions (i.e., exceptions that are not part of the process model). Exception handlers, in this case, are constructed dynamically from a basic set of protocols. The approach seems promising, although some concerns related to scalability can be identified and no integration within any agent platform is discussed.

(Gutierrez-Garcia et al., 2009) proposed an approach in which exception handlers are modeled through obligations in deontic logic. Exceptions are seen as abnormal situations in which agents cannot release an obligation. Exception handlers are modeled in terms of new obligations to be issued. Despite both approaches exploit obligations as a mechanism to deliver exception handling, this approach differs from ours because: (i) it is not framed in an organizational dimension; (ii) exceptions are not first-class objects, constituting a feedback for an exceptional situation, but are rather simply conceived as abnormal situations emerging during the enactment of an interaction protocol. At the same time, exception handling is not conceived in terms of responsibilities taken by the agents, but rather in terms of duties that fall on the agents. Finally, the proposal remains mainly theoretical, with no support from a software engineering perspective.

SARL (Rodriguez et al., 2014) recently adopted an exception handling mechanism based on the definition of special classes of events, amounting to exceptions. Agents have also the possibility to propagate failure events to their parent agent in the holarchy. The proposal is strongly inspired by the approach adopted in Akka, where actors report exceptions to their parents, which in turn must enact suitable supervision strategies to handle them. In SARL, however, autonomous agents do not explicitly take responsibility within the society for raising and handling exception. This is a major limitation because it hinders the creation of social expectations on the agents' behavior w.r.t. exception handling.

BPMN, for its part, provides a fairly expressive mechanism to model exceptional situations possibly occurring during the distributed execution of a business process.

	Autonomy Preservation	Decentralization	Responsibility Distribution	Reliable Feedback	Platform Integration
Programming languages	-	-	✓	✓	✓
Supervision in Akka	-	✓	✓	✓	✓
CA Actions	-	✓	✓	✓	✗
Error Events in BPMN	-	✓	✓	✓	✓
Guardian	✗	✗	✗	✗	✓
Sentinels	✗	✗	✗	✓	✗
SaGE	✓	✓	✗	✗	✓
Platon et al.	✓	✗	✗	✗	✓
Mallya and Singh	✓	✓	✓	✗	✗
Gutierrez-Garcia et al.	✓	✓	✗	✗	✗
SARL	✓	✓	✗	✓	✓
Rainbow	✗	✗	✗	✓	✓
Jason Contingency Plans	✓	✗	✗	✗	✓
Our Proposal	✓	✓	✓	✓	✓

Table 6.1: Feature comparison of the most prominent exception handling approaches.

In this sense, our proposal paves the way for a fruitful application of MAS to support the realization of distributed business processes.

In the context self-adaptive system, the approach adopted in the Rainbow architecture (Garlan et al., 2009) bears some analogies with concept of recovery strategy in our proposal. However, the two approaches are substantially different in nature. In (Garlan et al., 2009), the whole adaptation process is carried out by the components constituting the proposed architecture, that is, it is not part of the system at hand. In our approach, on the contrary, exception handling is embodied into the agents, leveraging their distributed nature. At the same tame, Rainbow does not take into account the autonomy of system's components. For this reason, no explicit responsibility distribution among them is foreseen for reporting and handling adaptation conditions. In our perspective, instead, such a responsibility distribution is fundamental since it enables the establishment of social relationships among autonomous components, providing expectations on their behavior concerning exceptional situations.

Table 6.1 summarizes the main differences between the cited approaches and ours with respect to a set of features that, we believe, should be exhibited by an exception handling mechanism to be suitable for a MAS setting.

Autonomy Preservation. To be suitable for use in MAS, an exception handling mechanism should not interfere with the agents' autonomy. Our proposal respects this requirement. An explicit responsibility assumption allows to create expectation on the agents' behavior w.r.t. exception handling. Nonetheless, agents remain completely free to violate the obligations issued towards them, despite being possibly sanctioned. At the same time, agents are autonomous in deliberating the most suitable way to carry out their assigned tasks, either amounting to raising or handling exceptions.

Decentralization. Multi-agent systems provide the abstractions to model and build distributed and heterogeneous systems in a simple and straightforward way, in order to manage their complexity. The proposed mechanism leverages this distributed

nature in the exception handling process, as well. Exceptions are raised and handled in synergy by the society of agents taking part in the organization. Each agent, is conceived independently from the others and the organization coordinates the distributed execution. At the same time, the exception handling mechanism is seamlessly integrated within the organizational infrastructure. Such an infrastructure is reified in the environment where the agents are situated in a distributed way, too.

Responsibility Distribution. As pointed out by Goodenough, exception handling is to be seen as a mechanism to increase the generality and facilitate the composition of operations. Responsibility distribution then becomes fundamental to determine which component is entitled to report the occurrence of an exception to whom and which one is entitled to handle it. This is the foundation of our proposed exception handling mechanism which aims at creating, through a properly devised set of responsibilities, a bridge between the agents in charge of raising exceptions and the ones in charge for their handling. At the same time, the approach allows to move the responsibility for handling the exception outside of the failing agent, increasing the generality of the applied recovery. This aspect justifies the adoption of the organization metaphor as a particularly suitable tool to deliver an exception handling mechanism. At the same time, it enables a uniform agent programming approach, where each agent must be equipped with the means to fulfill its responsibilities, should they concern exception handling or not.

Reliable Feedback. The availability of a feedback passed from the component raising the exception to the component handling it is the second pillar of Goodenough's vision. Indeed, a reliable feedback coming from an informed source allows to increase the situational awareness of the perturbation, with straightforward benefits in its handling. This is especially true in a multi-agent setting, where each agent may have a different and partial view of the environment and of the overall ongoing execution. Our proposal systematize the way in which this relevant information is produced, encoded, delivered, and exploited for recovery.

Platform Integration. Together with the conceptual and theoretical soundness, we believe that the presence of a concrete programming support is fundamental for any concrete application of exception handling. For this reason, we decided to implement the proposed model in the context of the JaCaMo framework, even if it is conceptually independent from any specific platform. The resulting solutions proved to be effective in dealing with a wide range of situations, in multiple application scenarios.

7 Discussion and Future Directions

When a system meets a perturbation it needs, in a way, to reconfigure. To this aim: (i) relevant information should be asked to informed agents, and delivered by these in the right format, (ii) agents should be supplied with the means for understanding who is entitled to ask what to whom, and under which circumstances, (iii) an appropriate handling, based on the available information, must be applied. All such things cannot be improvised, and one would not want the agents to start negotiations to build the new configuration at the moment because that may be an obstacle to a prompt answer.

Exception handling serves this purpose effectively, by systematizing the way in which channels are open, through which relevant feedback concerning the perturbation can reach the right component for handling it. The problem of building feedback frameworks, crucial for obtaining robustness, brings out the closeness between exception handling and the notion of *accountability*.

Accountability and responsibility are sibling concepts, often used interchangeably. However, they are different in nature. The Cambridge Dictionary defines accountability as:

> *The fact of being responsible for what you do and able to give a satisfactory reason for it, or the degree to which this happens.*

Under this perspective, we believe that accountability can be a useful tool for the realization of agent organizations that exhibit robustness as a design property, as well.

7.1 Exception Handling and Accountability

The notion of *accountability* has recently gained the attention of many authors in the MAS field (see e.g., (Chopra and Singh, 2016; Baldoni et al., 2018a; Cranefield et al., 2018; Baldoni et al., 2019)), who see a powerful software engineering tool in it. We agree with this view, and add that accountability can have a fundamental role in the design and realization of robust agent systems.

In particular, we believe that exception handling can be effectively conceived, more generally, in terms of accountability relationships among agents in an organization. In the following we briefly introduce the notion accountability and its usage in human organizations. We then illustrate how our proposed exception handling model can be read back as an accountability mechanism.

7.1.1 Accountability in the Human World

Accountability is a well-known concept in sociology and in human organizations. It typically "emerges as a primary characteristic of governance where there is a sense of agreement and certainty about the legitimacy of expectations between the community members." (Dubnick and Justice, 2004). This makes accountability a mechanism and instrument of administrative and political power, through which organizations can ensure the compliance of their processes to predefined standards, as well as the force that enables changes aimed at improving the organization (Bovens, 2010).

In general, accountability (Garfinkel, 1967; Dubnick and Justice, 2004; Grant and Keohane, 2005; Baldoni et al., 2016; Baldoni et al., 2019) can be seen as the assumption of responsibility for decisions and actions that an individual, or an organization, has towards another party. The term accountability has its roots in Latin, where it is related to the verb *computare*, to compute or calculate. Roughly speaking, an accountable person has the capability to provide an *account* about a condition of interest (Dubnick, 2014); that is, a person can be accountable for a

condition, only if she has some competence, or knowledge, about the very same condition.

In many cultures, accountability is associated with blame (Dubnick, 2013), either *post factum* (who is to blame for an act or an error that has occurred), or *pre factum* (who is blameworthy for errors not yet occurred), but this is a very partial view that disregards the potential involved in relationships concerning the ability and the designation to provide response about something to someone who is legitimated to ask. In sociology, and in ethnomethodology in particular, it is seen as a basic mechanism that allows individuals to constitute societies (Garfinkel, 1967; Rawls, 2008). In political sciences (Grant and Keohane, 2005), accountability is a relationship between a power-wielder and those holding them accountable. It expresses a general recognition of the *legitimacy of the authority of the parties* that are involved in the relationship: one to exercise particular powers and the other to hold them to provide an account. In the literature, the party who is legitimately required to provide the account is commonly called "account giver", or *a-giver*, while the party who can legitimately ask, under some agreed conditions, to the other party an account about a process of interest is called the "account taker", or *a-taker*, (Chopra and Singh, 2014; Baldoni et al., 2018a; Cranefield et al., 2018).

Consequently, it is possible to recognize in accountability two main dimensions:

1. *Normative dimension* (expectation), capturing the legitimacy of asking and the availability to provide accounts, yielding expectations on the individuals' behavior;

2. *Structural dimension* (control), capturing that, for being accountable about a process, an individual must have control over that process and have awareness of the situation she will account for.

Control often is interpreted as the ability to bring about events, possibly through other agents (see e.g., (Marengo et al., 2011; Yazdanpanah and Dastani, 2016)), that is, to have power over a situation of interest. In the case of accountability, this means that a-takers should be able to build the account themselves, either because

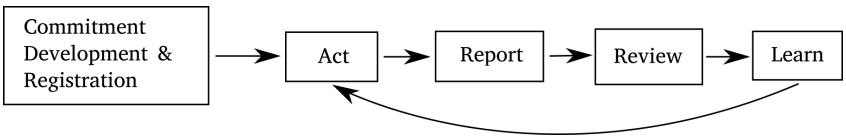

Figure 7.1: Accountability frameworks in human organizations.

they were directly involved in the attempt of bringing about some event, or because they can get the information that is necessary to build an account through other individuals.

Many organizations, and international agencies (see e.g., (Sustainable Energy for All Initiative, n.d.; Executive Board of the United Nations Development Programme and of the United Nations Population Fund, 2008; United Nations Children's Fund, 2009; Zahran, 2011; World Health Organization, 2015; Office of the Auditor General of Canada, 2002)), recognize accountability as a key component for their proper functioning. Accountability is, in fact, the mechanism through which important properties (such as trust, transparency, and robustness, just to mention some), can be established within a human organization.

Accountability frameworks, like (Executive Board of the United Nations Development Programme and of the United Nations Population Fund, 2008), are organization-wide processes for monitoring, analyzing, and improving performance in all aspects of the organization, based on actual data, Figure 7.1[11]. The figure draws a pretty general schema showing the loop that goes from decisions, and actions through report to learning, a term that is used here to capture the modification of the organization itself aimed at bettering its performance, based on the gathered accounts of those who were involved. It is worth noting that this process is not an end in itself, but its ultimate goal is to exploit the information obtained through the reporting activity to learn how the whole organizational structure can be modified and improved w.r.t. its objectives in a virtuous circle.

[11] The picture is inspired by the framework schemas described in (Sustainable Energy for All Initiative, n.d.; Zahran, 2011).

Indeed, failure, or partial achievement of the desired results, needs to be understood in order to either modify the organizational goals (when they turn out to be unreachable in that context with those resources), or to modify the organization itself, its practices, its structure, its competences, in order to improve performance. Accountability is supported by formally documented functions, responsibilities, authority, policies and gives managers the means to address recurring and systemic issues, and to incorporate lessons learned into future activities.

Accounts are used also by external bodies, with an oversight function, with the aim of verifying the adherence of behaviour to specific standards – see, for instance the General Data Protection Regulation by the European Community[12].

Although accountability frameworks vary considerably, depending on the kind of actors that are involved, on the kind of commitments, and on the activities that may be put under scrutiny, the same approach can be seen in most (human) organizations: the accountability framework provides the infrastructure that is necessary to a body made of many offices and individuals, geographically distributed, to collect information and provide it to those who are competent to interpret it, to take decisions and influence the future activity of the whole organization.

7.1.2 Conceiving Exception Handling as Accountability

Broadly speaking, accountability relationships among principals in a distributed system allow to open channels through which relevant local information concerning the execution of a process (i.e., accounts) flow from informed sources (a-givers) to those agents which are entitled to ask for it and competent to understand the answer (a-takers). In this sense, accountability supports robustness when the account (feedback) about a perturbation is reported to the agent who is responsible for treating that perturbation, similarly to what happens in exception handling mechanisms, as explained in the previous chapters. So, by enriching the specification of an organization with a proper set of accountability relationships among the agents,

[12]GDPR `https://eur-lex.europa.eu/eli/reg/2016/679/oj`.

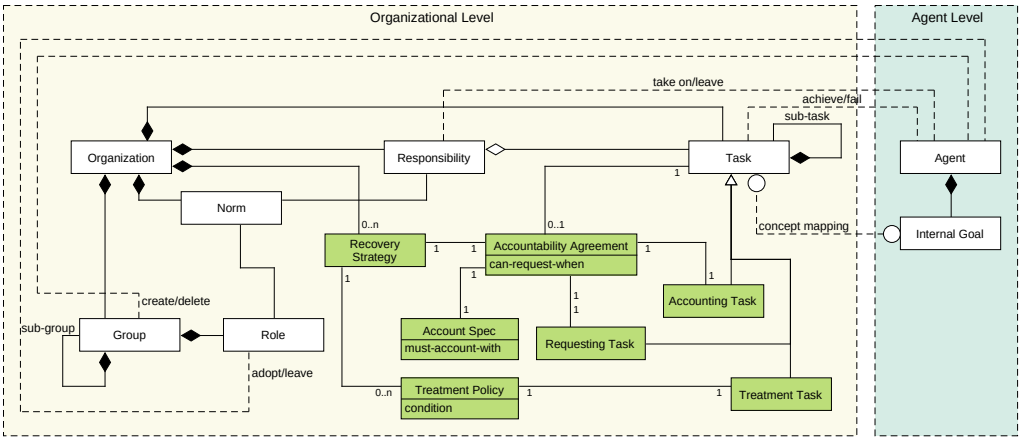

Figure 7.2: Our proposed conceptual model of exception handling in multi-agent organizations read in terms of accountability.

a designer can capture how the relevant information concerning exceptional events is to be propagated along the organizational structure.

Even though we focus on exceptional situations, accountability enables the treatment of accounts concerning behavior in positive cases, too; a feature that supports, e.g., compliance to standards and transparency, as well as the integration of oversight frameworks (Baldoni et al., 2020a; Baldoni et al., 2020b).

Figure 7.2 shows the abstract model presented in Chapter 4, where the concepts related to exception handling are interpreted in terms of accountability. In particular, the concept of `Notification Policy` is mapped to `Accountability Agreement`, the `Exception Spec` amounts to the concept of `Account Spec`, and the `Handling Policy` is renamed `Treatment Policy`. `Throwing Tasks` and `Catching Tasks` are mapped, respectively, to the concepts of `Accounting Task` and `Treatment Task`. Additionally, another type of task – the `Requesting Task` – is introduced, and associated with the concept of accountability agreement.

Accountability agreements generalize the notion of notification policy in our exception handling model. In particular, a notification policy encodes, how and when an exception has to be raised during the functioning of an organization, upon the occurrence of a perturbation. In other words, it captures when an account concerning

the perturbation (i.e., the exception) must be produced by the agent responsible for the corresponding throwing goal. Following a similar perspective, an accountability agreement captures the permission for an agent to request (through a requesting task) an account about the state of a given task when some conditions hold, and the obligation for another to provide such account (through an accounting task) when requested.

Notification policies represent a special class of accountability agreements, in which the account for a perturbation is requested automatically every time the perturbation occurs, thereby constraining the way in which agents produce and consume accounts. For this reason, in the particular case, we can omit the requesting task and the notification policy results in an obligation to produce the account (i.e., raise the exception) as soon as the condition that triggers the policy (encoding the perturbation of interest) holds. Similarly, exceptions are special kinds of accounts, which concern the occurrence of perturbations. Recovery strategies become then a way to connect, through a treatment policy, the account for a perturbation with the treatment task that can properly tackle it – in a way that is oriented towards recovery.

As before, the association between accountability agreement and task captures the object of the account. That is, the a-giver is expected to produce an account that objectively concern the task indicated via this association. The same association was present between a notification policy and a task, representing the target of the policy.

An accountability agreement is characterized by a `can-request-when` attribute (mapping the condition of interest), that specifies when an account request is legitimate, i.e., when the a-taker has the permission of asking for an account. In exception handling, this condition amounts to the perturbation at hand. Such a condition represents the circumstance in which the a-giver can be held to account. When such a condition is not met, the a-giver is not obliged to produce the account. For instance, a buyer may hold a seller to account for some goods, but the seller will have to provide a feedback only if the purchase actually occurred, that is, only if the

payment took place. Here, payment is the contextual condition that gives the buyer the right to request the account.

The association between accountability agreement and requesting task specifies who will serve the purpose of being a-taker, that is, the one who is responsible for the requesting task. Note that, in some cases, an account request is the result of an internal deliberation of the a-taker; this is a more general approach w.r.t. what expressed by notification policies in the exception handling model. In that case, in fact, the request phase was omitted because we assumed accounts to be needed for any perturbation deemed to possibly occur.

The concept of account spec, characterized by a `must-account-with` attribute, captures the type of knowledge that the a-giver feeds back to the a-taker upon request. This is comparable with the informational content encoded by the exception spec.

The association between accountability agreement and accounting task specifies who will serve the purpose of being a-giver, that is, the one who is responsible for the accounting task.

Finally, we highlight that accountability agreements, just like notification policies, bring along normative expectations that can be formulated according to the normative layer (`Norm` in the figure).

From a computational perspective, in (Chopra and Singh, 2014; Chopra and Singh, 2016) the authors explain how, within Socio-Technical Systems, accountability plays a fundamental role in balancing the principals' autonomy: a principal can decide to violate any expectation for which it is accountable, however, by way of accountability the principal would be held to account about that violation.

The proposal in (Cranefield et al., 2018) recognizes the value of accountability in the development of software and makes a proposal that is complementary to ours. Specifically, the authors focus on the issue of answer production in presence of an accountability relationship, a problem that involves: how to properly define the temporal window to consider? Which pieces of information are relevant and, thus,

are to be kept in this temporal interval? Which questions are suitable to be asked in this setting? The account giving agent produces an answer in terms of its internal mechanisms. What that proposal does not provide is the organizational view of the system of interacting agents and they do not tackle robustness and exceptions.

In (Chopra and Singh, 2018), accountability enables the process of norms adaptation by feeding outcomes back into the design-phase. In this approach, the account is a justification of an agent's norm-violating behavior. This is a different understanding of accounts than ours because, in our approach, account givers are not rule violators: they meet perturbations, and provide information about the encountered situations. The account takers, on their hand, will interpret the received accounts – possibly combining them with further information provided by other agents or that simply belongs to the callee's level. The adaptation process in (Chopra and Singh, 2018), that consists in norm modification, however, can be seen as a kind of robustness. Our objective is different: we do not target norm modification, but the achievement of the organizational goal despite the occurrence of perturbations. The two approaches are not in contrast, rather, they complement each other. They are both exemplifications of the perspective put forward in (Alderson and Doyle, 2010), for which a property of a system is robust if it is invariant with respect to a set of perturbations. The difference lies in the type of perturbations the two approaches aim at.

MOCA (Baldoni et al., 2019) provides an information model of accountability, that captures the kind of facts that must be available to allow the identification of account givers in certain situation of interest. The model is given in Object-Role Modeling (ORM) (Halpin and Morgan, 2008) due to the relational nature of the represented concepts, and enables automatic verification of consistency. The information model is centered around two basic concepts: *just expectation* and *control*. Just expectation is intended as the mutual awareness and acceptance of an accountability relationship between the involved a-giver and a-taker. Control, instead, is intended as the power, possibly exerted indirectly by means of other agents, of achieving a condition of interest.

7.2 Conclusion and Future Directions

In conclusion, in this thesis we have presented an exception handling mechanism for use in multi-agent systems, grounded on the notions of responsibility and feedback. Its main purpose is to increase system robustness while preserving, at the same time, autonomy of the components (agents). Exception handling, indeed, emerges from the need for conjugating robustness with modularity among software components, and MAS bring this feature to an extreme.

To reach the objective, we proposed to leverage the concept of responsibility in multi-agent organizations not only to model the duties of the agents in relation to the organizational goal, but also to enable the realization of mechanisms for raising and handling exceptions that may occur within the organization operation.

Agents joining an organization are required to explicitly take on the responsibilities for providing feedback about the context where exceptions are detected, and for handling these exceptions as soon the feedback is available. In this way, the normative system, which coordinates the agents' fulfillment of their responsibilities, becomes a tool to specify and govern both the correct and exceptional behavior of the system, uniformly.

The proposal has been concretely set and evaluated in the JaCaMo multi-agent platform by means of a set of use cases. An interesting future development includes the integration of the approach in other agent platforms, such as SARL (Rodriguez et al., 2014), JADE (Bellifemine et al., 1999), or ASTRA (Collier et al., 2015). These languages do not typically encompass an organizational dimension and thereby require the deployment of other mechanisms to coordinate the process of responsibility distribution. In this sense commitment-based approaches, such as (Singh, 1999; Baldoni et al., 2014), may provide useful insights. At the same time, it would be useful to determine whether and how the architectural constraints of the cited platforms may affect and be exploited to accommodate exception handling effectively.

Another exciting research direction concerns the handling of *unexpected exceptions*, which are not defined "by contract" at design time. Instead, they emerge at runtime

and for this reason the responsibility to raise and handle them cannot be distributed *a priori*. Under this perspective, the research field of self-adaptive systems could provide useful insights.

Still, as explained above, exception handling can be conceived, more generally, as an accountability mechanism. Another promising direction for future work would be an extension of the presented programming platform in terms of accountability, as well. Accountability indeed, has the potential to support, besides robustness, a wide range of non-functional requirements, such as transparency, auditability, explainability, adaptability, and innovation. A framework allowing system components (agents in our perspective), to exchange information in a structured way, at a different level of that of the outcomes that are specified by functional requirements, can set the ground for capturing a wide range of such non-functional requirements.

An accountability platform could provide evidence of agent conduct in a transparent and automated way, while simultaneously simplifying the system reconfiguration in case of perturbations. Potential applications range widely in such diverse fields as, just to name a few, resource management, smart cities, industry 4.0, business process management, public administration, and decision support.

References

Alderson, D. L. and Doyle, J. C. (2010). "Contrasting Views of Complexity and Their Implications for Network-Centric Infrastructures". In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 40.4, pp. 839–852 (cit. on pp. 1, 3, 12, 13, 157).

Aldewereld, H., O. Boissier, V. Dignum, P. Noriega, and J. Padget, eds. (2016). *Social Coordination Frameworks for Social Technical Systems*. Vol. 30. Law, Governance and Technology Series. Springer (cit. on p. 59).

Baldoni, M., Baroglio, C., Boissier, O., May, K. M., Micalizio, R., and Tedeschi, S. (2018a). "Accountability and Responsibility in Agents Organizations". In: *PRIMA 2018: Principles and Practice of Multi-Agent Systems, 21st International Conference*. Vol. 11224. Lecture Notes in Computer Science. Springer, pp. 403–419 (cit. on pp. 150, 151).

Baldoni, M., Baroglio, C., and Capuzzimati, F. (2014). "A Commitment-Based Infrastructure for Programming Socio-Technical Systems". In: *ACM Transactions on Internet Technology* 14.4, pp. 1–23 (cit. on p. 158).

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and Tedeschi, S. (2016). "Computational Accountability". In: *Proceedings of the AI*IA Workshop on Deep Understanding and Reasoning: A Challenge for Next-generation Intelligent Agents 2016 co-located with 15th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2016), Genova, Italy, November 28th, 2016*. Vol. 1802. CEUR Workshop Proceedings. CEUR-WS.org, pp. 56–62 (cit. on pp. 8, 150).

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and Tedeschi, S. (2018b). "Computational Accountability in MAS Organizations with ADOPT". In: *Applied Sciences* 8.4 (cit. on pp. 6, 61).

Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., and Tedeschi, S. (2019). "MOCA: An ORM MOdel for Computational Accountability". In: *Journal of Intelligenza Artificiale* 13.1, pp. 5–20 (cit. on pp. 8, 150, 157).

Baldoni, M., Baroglio, C., and Micalizio, R. (2020a). "Fragility and Robustness in Multiagent Systems". In: *Engineering Multi-Agent Systems*. Vol. 12589. Lecture Notes in Computer Science. Springer, pp. 61–77 (cit. on p. 154).

Baldoni, M., Baroglio, C., Micalizio, R., and Tedeschi, S. (2020b). "Is Explanation the Real Key Factor for Innovation?" In: *Proceedings of the Italian Workshop on Explainable Artificial Intelligence co-located with 19th International Conference of the Italian Association for Artificial Intelligence, XAI.it@AIxIA 2020, Online Event, November 25-26, 2020*. Vol. 2742. CEUR Workshop Proceedings. CEUR-WS.org, pp. 87–95 (cit. on p. 154).

Bellifemine, F., Poggi, A., and Rimassa, G. (1999). “JADE - A FIPA-compliant agent framework”. In: *Proceedings of the Practical Applications of Intelligent Agents* (cit. on pp. 2, 158).

Boella, G., Torre, L. W. N. van der, and Verhagen, H. (2008). “Introduction to the special issue on normative multiagent systems”. In: *Autonomous Agents and Multi-Agent Systems* 17.1, pp. 1–10 (cit. on pp. 6, 59).

Boella, G., Van Der Torre, L., and Verhagen, H. (2006). “Introduction to normative multiagent systems”. In: *Computational & Mathematical Organization Theory* 12.2-3, pp. 71–79 (cit. on pp. 6, 59).

Boer, F. S. de, Hindriks, K. V., Hoek, W. van der, and Meyer, J.-J. C. (2007). “A verification framework for agent programming with declarative goals”. In: *Journal of Applied Logic* 5.2, pp. 277–302 (cit. on p. 38).

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). “Multi-agent Oriented Programming with JaCaMo”. In: *Science of Computer Programming* 78.6, pp. 747–761 (cit. on pp. 7, 57, 59, 72, 76, 103).

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons (cit. on pp. 2, 36–38, 49, 50, 72).

Bovens, M. (2010). “Two Concepts of Accountability: Accountability as a Virtue and as a Mechanism”. In: *West European Politics* 33.5, pp. 946–967 (cit. on p. 150).

Bratman, M. E. (1987). *Intention, plans, and practical reason*. Harvard University Press (cit. on p. 37).

Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988). “Plans and resource-bounded practical reasoning”. In: *Computational Intelligence* 4.3, pp. 349–355 (cit. on p. 37).

Brito, M. de, Hübner, J. F., and Boissier, O. (2017). “Situated artificial institutions: stability, consistency, and flexibility in the regulation of agent societies”. In: *Autonomous Agents and Multi-Agent Systems*, pp. 1–33 (cit. on p. 57).

Buhr, P. A. and Mok, W. Y. R. (2000). “Advanced Exception Handling Mechanisms”. In: *IEEE Transactions on Software Engineering* 26.9, pp. 820–836 (cit. on pp. 2, 13).

Campbell, R. H. and Randell, B. (1986). “Error recovery in asynchronous systems”. In: *IEEE Transactions on Software Engineering* 8, pp. 811–826 (cit. on pp. 27, 43).

Cheng, B. H. C., Lemos, R. de, Giese, H., et al. (2009). “Software Engineering for Self-Adaptive Systems: A Research Roadmap”. In: *Software Engineering for Self-Adaptive Systems*. Vol. 5525. Lecture Notes in Computer Science. Springer, pp. 1–26 (cit. on p. 32).

Chopra, A. K. and Singh, M. P. (2014). “The thing itself speaks: Accountability as a foundation for requirements in sociotechnical systems”. In: *2014 IEEE 7th International Workshop on Requirements Engineering and Law (RELAW)*, pp. 22–22 (cit. on pp. 8, 151, 156).

Chopra, A. K. and Singh, M. P. (2016). “From social machines to social protocols: Software engineering foundations for sociotechnical systems”. In: *Proceedings of the 25th International Conference on World Wide Web*, pp. 903–914 (cit. on pp. 6, 60, 150, 156).

Chopra, A. K. and Singh, M. P. (2018). “Sociotechnical Systems and Ethics in the Large”. In: *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society, AIES 2018, New Orleans, LA, USA, February 02-03, 2018*. ACM, pp. 48–53 (cit. on p. 157).

Collier, R. W., Russell, S., and Lillis, D. (2015). “Reflecting on agent programming with AgentSpeak(L)”. In: *PRIMA 2015: Principles and Practice of Multi-Agent Systems*. Vol. 9387. Lecture Notes in Computer Science. Springer, pp. 351–366 (cit. on pp. 38, 158).

Corkill, D. D. and Lesser, V. R. (1983). “The Use of Meta-Level Control for Coordination in Distributed Problem Solving Network”. In: *Proceedings of the 8th International Joint Conference on Artificial Intelligence (IJCAI’83)*. William Kaufmann, pp. 748–756 (cit. on pp. 5, 58).

Cossentino, M., Lopes, S., and Sabatucci, L. (2020). “A Tool for the Automatic Generation of MOISE Organisations From BPMN”. In: *Proceedings of the Workshop on 21st Workshop "From Objects to Agents", Bologna, Italy, September 14-16, 2020*. Vol. 2706. CEUR Workshop Proceedings. CEUR-WS.org, pp. 69–82 (cit. on p. 120).

Cossentino, M., Lopes, S., and Sabatucci, L. (2021). “Automatic Definition of MOISE Organizations for Adaptive Workflows”. In: *Proceedings of the 13th International Conference on Agents and Artificial Intelligence, ICAART 2021, Volume 1, Online Streaming, February 4-6, 2021*. SciTePress, pp. 125–136 (cit. on p. 120).

Cranefield, S., Oren, N., and Vasconcelos, W. W. (2018). “Accountability for Practical Reasoning Agents”. In: *Agreement Technologies - 6th International Conference, AT 2018, Bergen, Norway, December 6-7, 2018, Revised Selected Papers*. Vol. 11327. Lecture Notes in Computer Science. Springer, pp. 33–48 (cit. on pp. 150, 151, 156).

Cristian, F. (1985). “Exception handling and software fault tolerance”. In: *Reliable Computer Systems*. Springer, pp. 154–172 (cit. on pp. 2, 13).

Dastani, M. (2008). “2APL: a practical agent programming language”. In: *Autonomous Agents and Multi-Agent Systems* 16.3, pp. 214–248 (cit. on p. 38).

Dastani, M., Tinnemeier, N. A. M., and Meyer, J.-J. C. (2009). “A programming language for normative multi-agent systems”. In: *Handbook of Research on Multi-Agent Systems: semantics and dynamics of organizational models*. IGI Global, pp. 397–417 (cit. on pp. 5, 59).

Dellarocas, C. and Klein, M. (2000). “An experimental evaluation of domain-independent fault handling services in open multi-agent systems”. In: *Proceedings Fourth International Conference on MultiAgent Systems*. IEEE, pp. 95–102 (cit. on pp. 41, 142).

Dignum, V. (2004). “A model for organizational interaction: based on agents, founded in logic”. Published by SIKS. PhD thesis. Utrecht University, The Netherlands (cit. on p. 59).

Dignum, V. (2009). *Handbook of Research on Multi-agent Systems: Semantics and Dynamics of Organizational Models* (cit. on p. 58).

Dignum, V., Dignum, F., and Meyer, J.-J. C. (2004a). “An agent-mediated approach to the support of knowledge sharing in organizations”. In: *The Knowledge Engineering Review* 19.2, pp. 147–174 (cit. on pp. 5, 57, 58).

Dignum, V., Vázquez-Salceda, J., and Dignum, F. (2004b). "OMNI: Introducing Social Structure, Norms and Ontologies into Agent Organizations". In: *Programming Multi-Agent Systems, Second International Workshop ProMAS, Selected Revised and Invited Papers*. Vol. 3346. Lecture Notes in Computer Science. Springer, pp. 181–198 (cit. on pp. 5, 57, 59, 60).

Dubnick, M. J. (2013). *Blameworthiness, Trustworthiness, and the Second-Personal Standpoint: Foundations for an Ethical Theory of Accountability*. Presented at EGPA Annual Conference, Group VII: Quality and Integrity of Governance, Edinburgh, Scotland (cit. on pp. 8, 151).

Dubnick, M. J. (2014). "Accountability as a Cultural Keyword". In: *Oxford Handbook on Public Accountability*. Oxford University Press, pp. 23–38 (cit. on p. 150).

Dubnick, M. J. and Justice, J. B. (2004). *Accounting for Accountability*. Annual Meeting of the American Political Science Association (cit. on pp. 8, 150).

Elder-Vass, D. (2011). *The Causal Power of Social Structures: Emergence, Structure and Agency*. Cambridge University Press (cit. on pp. 3, 58).

Esteva, M., Rodríguez-Aguilar, J.-A., Sierra, C., Garcia, P., and Arcos, J. L. (2001). "On the Formal Specification of Electronic Institutions". In: *Agent Mediated Electronic Commerce: The European AgentLink Perspective*. Vol. 1991. Lecture Notes in Computer Science. Springer, pp. 126–147 (cit. on p. 59).

Executive Board of the United Nations Development Programme and of the United Nations Population Fund (2008). *The UNDP accountability system, Accountability framework and oversight policy*. Tech. rep. DP/2008/16/Rev.1. United Nations (cit. on p. 152).

Feltus, C. (2014). "Aligning Access Rights to Governance Needs with the Responsability MetaModel (ReMMo) in the Frame of Enterprise Architecture". PhD thesis. University of Namur, Belgium (cit. on pp. 7, 57, 60).

Fernandez, J.-C., Mounier, L., and Pachon, C. (2005). "A Model-based Approach for Robustness Testing". In: *Testing of Communicating Systems*. Vol. 3502. Lecture Notes in Computer Science. Springer, pp. 333–348 (cit. on pp. 1, 12).

Fernández-Díaz, Á., Benac-Earle, C., and Fredlund, L.-A. (2015). "Adding distribution and fault tolerance to Jason". In: *Science of Computer Programming* 98. Special Issue on Programming Based on Actors, Agents and Decentralized Control, pp. 205–232 (cit. on p. 51).

Fischer, K., Schillo, M., and Siekmann, J. (2003). "Holonic multiagent systems: A foundation for the organisation of multiagent systems". In: *Holonic and Multi-Agent Systems for Manufacturing*. Vol. 2744. Lecture Notes in Computer Science. Springer, pp. 71–80 (cit. on p. 48).

Fornara, N., Viganò, F., Verdicchio, M., and Colombetti, M. (2008). "Artificial institutions: a model of institutional reality for open multiagent systems". In: *Artificial Intelligence and Law* 16.1, pp. 89–105 (cit. on pp. 5, 59).

Friedman, D. P., Haynes, C. T., and Kohlbecker, E. (1984). "Programming with Continuations". In: *Program Transformation and Programming Environments*. Springer, pp. 263–274 (cit. on p. 19).

Garfinkel, H. (1967). *Studies in ethnomethodology*. Prentice-Hall Inc. (cit. on pp. 8, 150, 151).

Garlan, D., Schmerl, B., and Cheng, S.-W. (2009). "Software Architecture-Based Self-Adaptation". In: *Autonomic Computing and Networking*. Springer, pp. 31–55 (cit. on pp. 32, 33, 145).

Georgeff, M. P. and Lansky, A. L. (1987). "Reactive reasoning and planning". In: *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 2*. Vol. 87. AAAI'87. AAAI, pp. 677–682 (cit. on p. 38).

Gerber, C., Siekmann, J., and Vierke, G. (1999). *Holonic multi-agent systems*. Tech. rep. Deutsches Forschungszentrum für Künstliche Intelligenz GmbH (cit. on p. 48).

Goodenough, J. B. (1975a). "Exception Handling Design Issues". In: *ACM SIGPLAN Notices* 10.7, pp. 41–45 (cit. on pp. 2, 13, 14).

Goodenough, J. B. (1975b). "Exception Handling: Issues and a Proposed Notation". In: *Communications of the ACM* 18.12, pp. 683–696 (cit. on p. 14).

Goodenough, J. B. (1975c). "Structured Exception Handling". In: *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '75. ACM, pp. 204–224 (cit. on p. 14).

Goodwin, J. (2015). *Learning Akka*. Packt Publishing Ltd (cit. on p. 22).

Grant, R. W. and Keohane, R. O. (2005). "Accountability and Abuses of Power in World Politics". In: *The American Political Science Review* 99.1, pp. 29–43 (cit. on pp. 8, 150, 151).

Gupta, M. (2012). *Akka essentials*. Packt Publishing Ltd (cit. on p. 21).

Gutierrez-Garcia, J. O., Koning, J.-L., and Ramos-Corchado, F. F. (2009). "An Obligation Approach for Exception Handling in Interaction Protocols". In: *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 03*. WI-IAT '09. IEEE Computer Society, pp. 497–500 (cit. on pp. 47, 143).

Hägg, S. (1997). "A sentinel approach to fault handling in multi-agent systems". In: *Multi-Agent Systems Methodologies and Applications*. Vol. 1286. Lecture Notes in Computer Science. Springer, pp. 181–195 (cit. on pp. 40, 41).

Halpin, T. and Morgan, T. (2008). *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers (cit. on p. 157).

Hewitt, C., Bishop, P., and Steiger, R. (1973). "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Morgan Kaufmann Publishers Inc., pp. 235–245 (cit. on p. 20).

Hindriks, K. V., De Boer, F. S., Hoek, W. van der, and Meyer, J.-J. C. (1999). "Agent programming in 3APL". In: *Autonomous Agents and Multi-Agent Systems* 2.4, pp. 357–401 (cit. on p. 38).

Hübner, J. F., Boissier, O., and Bordini, R. H. (2009). "A Normative Organisation Programming Language for Organisation Management Infrastructures". In: *Coordination, Organizations, Institutions and Norms in Agent Systems V*. Vol. 6069. Lecture Notes in Computer Science. Springer, pp. 114–129 (cit. on pp. 78, 79, 99).

Hübner, J. F., Boissier, O., and Bordini, R. H. (2010a). "From organisation specification to normative programming in multi-agent organisations". In: *Computational Logic in Multi-Agent Systems*. Vol. 6245. Lecture Notes in Computer Science. Springer, pp. 117–134 (cit. on p. 78).

Hübner, J. F., Boissier, O., and Bordini, R. H. (2011). "A normative programming language for multi-agent organisations". In: *Annals of Mathematics and Artificial Intelligence* 62.1, pp. 27–53 (cit. on pp. 78, 79).

Hübner, J. F., Boissier, O., Kitio, R., and Ricci, A. (2010b). "Instrumenting multi-agent organisations with organisational artifacts and agents". In: *Autonomous Agents and Multi-Agent Systems* 20.3, pp. 369–400 (cit. on p. 74).

Hübner, J. F., Sichman, J. S., and Boissier, O. (2007). "Developing Organised Multiagent Systems Using the MOISE+ Model: Programming Issues at the System and Agent Levels". In: *International Journal of Agent-Oriented Software Engineering* 1.3/4, pp. 370–395 (cit. on pp. 5, 57, 58, 60, 72).

An Architectural Blueprint for Autonomic Computing (2005). Tech. rep. IBM (cit. on pp. 32, 33).

"ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary" (2010). In: *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418 (cit. on pp. 11, 13).

Issarny, V. (2001). "Concurrent Exception Handling". In: *Advances in Exception Handling Techniques*. Vol. 2022. Lecture Notes in Computer Science. Springer, pp. 111–127 (cit. on p. 43).

Klein, M. and Dellarocas, C. (1999). "Exception handling in agent systems". In: *Proceedings of the Third Annual Conference on Autonomous Agents*. AGENTS '99. ACM, pp. 62–68 (cit. on pp. 41, 142).

Klein, M. and Dellarocas, C. (2000). "A knowledge-based approach to handling exceptions in workflow systems". In: *Computer Supported Cooperative Work (CSCW)* 9.3-4, pp. 399–412 (cit. on pp. 41, 142).

Klein, M., Rodriguez-Aguilar, J.-A., and Dellarocas, C. (2003). "Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death". In: *Autonomous Agents and Multi-Agent Systems* 7.1-2, pp. 179–189 (cit. on p. 42).

Krupitzer, C., Roth, F. M., VanSyckel, S., Schiele, G., and Becker, C. (2015). "A survey on engineering approaches for self-adaptive systems". In: *Pervasive and Mobile Computing* 17. 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian, pp. 184–206 (cit. on p. 34).

Lemos, R. de, Giese, H., Müller, H. A., et al. (2013). “Software Engineering for Self-Adaptive Systems: A Second Research Roadmap”. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Vol. 7475. Lecture Notes in Computer Science. Springer, pp. 1–32 (cit. on p. 32).

Lewerentz, C. and Lindner, T. (1995). *Formal development of reactive systems: case study production cell*. Vol. 891. Springer Science & Business Media (cit. on p. 134).

López y López, F. and Luck, M. (2003). “Modelling Norms for Autonomous Agents”. In: *4th Mexican International Conference on Computer Science (ENC 2003), 8-12 September 2003, Apizaco, Mexico*. IEEE Computer Society, pp. 238–245 (cit. on pp. 59, 60).

Macías-Escrivá, F. D., Haber, R., del Toro, R., and Hernandez, V. (2013). “Self-adaptive systems: A survey of current approaches, research challenges and applications”. In: *Expert Systems with Applications* 40.18, pp. 7267–7279 (cit. on pp. 32, 34).

Mallya, A. U. and Singh, M. P. (2005a). “A Semantic Approach for Designing Commitment Protocols”. In: *Agent Communication*. Vol. 3396. Lecture Notes in Computer Science. Springer, pp. 33–49 (cit. on p. 47).

Mallya, A. U. and Singh, M. P. (2005b). “Modeling Exceptions via Commitment Protocols”. In: *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS ’05. ACM, pp. 122–129 (cit. on pp. 46, 47, 142).

Mallya, A. U. (2005). “Modeling and Enacting Business Processes via Commitment Protocols among Agents”. PhD thesis (cit. on pp. 46, 142).

Marengo, E., Baldoni, M., Baroglio, C., Chopra, A., Patti, V., and Singh, M. (2011). “Commitments with regulations: reasoning about safety and control in REGULA”. In: *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. Vol. 2. IFAAMAS, pp. 467–474 (cit. on p. 151).

Meyer, B. (1988). *Object-oriented software construction*. Vol. 2. Prentice Hall New York (cit. on p. 7).

Miller, R. and Tripathi, A. (1997). “Issues with exception handling in object-oriented systems”. In: *ECOOP’97 — Object-Oriented Programming*. Vol. 1241. Lecture Notes in Computer Science. Springer, pp. 85–103 (cit. on p. 16).

Miller, R. and Tripathi, A. (2004). “The guardian model and primitives for exception handling in distributed systems”. In: *IEEE Transactions on Software Engineering* 30.12, pp. 1008–1022 (cit. on p. 39).

Object Management Group (2021). *BPMN Specification - Business Process Model and Notation*. Online, accessed 15/09/2021 (cit. on pp. 122, 130).

Office of the Auditor General of Canada (2002). *2002 December Report of the Auditor General of Canada: Chapter 9*. Accessed 15/09/2021. URL: `https://publications.gc.ca/collections/collection_2012/bvg-oag/FA1-2002-2-17-eng.pdf` (cit. on p. 152).

Omicini, A., Ricci, A., and Viroli, M. (2008). “Artifacts in the A&A Meta-Model for Multi-Agent Systems”. In: *Autonomous Agents and Multi-Agent Systems* 17.3, pp. 432–456 (cit. on p. 73).

Pereira, D. P. and Melo, A. C. V. de (2010). “Formalization of an architectural model for exception handling coordination based on CA action concepts”. In: *Science of Computer Programming* 75.5. Coordination Models, Languages and Applications (SAC’08), pp. 333–349 (cit. on p. 26).

Platon, E. (2007). “Modeling exception management in multi-agent systems”. PhD thesis. Université Pierre et Marie Curie, France (cit. on pp. 15, 44, 141).

Platon, E., Sabouret, N., and Honiden, S. (2007a). “A Definition of Exceptions in Agent-Oriented Computing”. In: *Engineering Societies in the Agents World VII*. Vol. 4457. Lecture Notes in Computer Science. Springer, pp. 161–174 (cit. on pp. 44, 141).

Platon, E., Sabouret, N., and Honiden, S. (2007b). “Challenges for Exception Handling in Multi-Agent Systems”. In: *Software Engineering for Multi-Agent Systems V*. Vol. 4408. Lecture Notes in Computer Science. Springer, pp. 41–56 (cit. on pp. 44, 141).

Platon, E., Sabouret, N., and Honiden, S. (2008). “An architecture for exception management in multiagent systems”. In: *International Journal of Agent-Oriented Software Engineering* 2.3, pp. 267–289 (cit. on pp. 44, 45, 50, 141).

Randell, B., Romanovsky, A., Stroud, R. J., Xu, J., and Zorzo, A. F. (1997). “Coordinated atomic actions: from concept to implementation”. In: *Submitted to Special Issue of IEEE Transactions on Computers* (cit. on p. 26).

Rao, A. S. (1996). “AgentSpeak(L): BDI agents speak out in a logical computable language”. In: *Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW ’96 Eindhoven, The Netherlands, January 22–25, 1996 Proceedings*. Vol. 1038. Lecture Notes in Computer Science. Springer, pp. 42–55 (cit. on pp. 38, 49, 73).

Rawls, A. W. (2008). “Harold Garfinkel, Ethnomethodology and Workplace Studies”. In: *Organization Studies* 29.701 (cit. on p. 151).

Reynolds, J. C. (1993). “The Discoveries of Continuations”. In: *Lisp and Symbolic Computation* 6.3-4, pp. 233–248 (cit. on p. 19).

Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009). “Environment Programming in CArtAgO”. In: *Multi-Agent Programming: Languages, Tools and Applications*. Springer, pp. 259–288 (cit. on pp. 2, 72).

Rodriguez, S., Gaud, N., and Galland, S. (2014). “SARL: A General-Purpose Agent-Oriented Programming Language”. In: *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. Vol. 3, pp. 103–110 (cit. on pp. 2, 48, 143, 158).

Romanovsky, A. (2001). “Coordinated Atomic Actions: How to Remain ACID in the Modern World”. In: *SIGSOFT Software Engineering Notes* 26.2, pp. 66–68 (cit. on p. 26).

Russell, S. and Norvig, P. (2002). *Artificial intelligence: a modern approach* (cit. on pp. 36, 37).

Sabatucci, L. and Cossentino, M. (2019). “Supporting Dynamic Workflows with Automatic Extraction of Goals from BPMN”. In: *ACM Transactions on Autonomous and Adaptive Systems* 14.2 (cit. on p. 120).

Sabatucci, L., Seidita, V., and Cossentino, M. (2018). “The Four Types of Self-adaptive Systems: A Metamodel”. In: *Intelligent Interactive Multimedia Systems and Services 2017*. Vol. 76. Smart Innovation, Systems and Technologies. Springer, pp. 440–450 (cit. on p. 34).

Schillo, M. and Fischer, K. (2002). “Holonic multiagent systems”. In: *Manufacturing Systems* 8.13, pp. 538–550 (cit. on p. 48).

Seborg, D. E., Mellichamp, D. A., Edgar, T. F., and Doyle III, F. J. (2010). *Process dynamics and control*. John Wiley & Sons (cit. on p. 32).

Shah, N., Chao, K.-M., Godwin, N., and James, A. (2005). “Exception diagnosis in open multi-agent systems”. In: *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. IEEE, pp. 483–486 (cit. on p. 42).

Shah, N., Chao, K.-M., Godwin, N., James, A., and Tasi, C.-F. (2006). “An empirical evaluation of a sentinel based approach to exception diagnosis in multi-agent systems”. In: *20th International Conference on Advanced Information Networking and Applications-Volume 1 (AINA'06)*. Vol. 1. IEEE, pp. 379–386 (cit. on p. 42).

Shah, N., Chao, K.-M., Godwin, N., Younas, M., and Laing, C. (2004). “Exception diagnosis in agent-based grid computing”. In: *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*. Vol. 4. IEEE, pp. 3213–3219 (cit. on p. 42).

Shoham, Y. (1993). “Agent-oriented programming”. In: *Artificial Intelligence* 60.1, pp. 51–92 (cit. on p. 38).

Silver, B. (2011). *BPMN Method and Style: With BPMN Implementer's Guide*. Cody-Cassidy Press (cit. on p. 29).

Singh, M. P. (1999). “An ontology for commitments in multiagent systems”. In: *Artificial Intelligence and Law* 7.1, pp. 97–113 (cit. on pp. 46, 158).

Singh, M. P. (2013). “Norms as a basis for governing sociotechnical systems”. In: *ACM Transactions on Intelligent Systems and Technology* 5.1, p. 21 (cit. on pp. 6, 59).

Sommerville, I. (2007). “Models for Responsibility Assignment”. In: *Responsibility and Dependable Systems*. Springer, pp. 165–186 (cit. on p. 7).

Sommerville, I., Storer, T., and Lock, R. (2009). “Responsibility modelling for civil emergency planning”. In: *Risk Management* 11.3, pp. 179–207 (cit. on p. 7).

Souchon, F. (2005). “SaGE, un Système de Gestion d'Exceptions pour la programmation orientée message: Le cas des Systèmes Multi-Agents et des Plates-formes à base de Composants Logiciels”. PhD thesis. Université des Sciences et Techniques du Languedoc, France (cit. on pp. 43, 142).

Souchon, F., Dony, C., Urtado, C., and Vauttier, S. (2003). “Improving exception handling in multi-agent systems”. In: *Software Engineering for Multi-Agent Systems II*. Vol. 2940. Lecture Notes in Computer Science. Springer, pp. 167–188 (cit. on pp. 43, 142).

Sustainable Energy for All Initiative (n.d.). *Accountability Framework*. Accessed 15/09/2021. URL: `https://sustainabledevelopment.un.org/content/documents/1644se4all.pdf` (cit. on p. 152).

Timm, I. J., Scholz, T., Herzog, O., Krempels, K., and Spaniol, O. (2006). “From Agents to Multiagent Systems”. In: *Multiagent Engineering, Theory and Applications in Enterprises*. Springer, pp. 35–51 (cit. on p. 3).

Tripathi, A. and Miller, R. (2001). “Exception Handling in Agent-Oriented Systems”. In: *Advances in Exception Handling Techniques*. Vol. 2022. Lecture Notes in Computer Science. Springer, pp. 128–146 (cit. on p. 39).

United Nations Children’s Fund (2009). *Report on the accountability system of UNICEF*. E/ICEF/2009/15, accessed 15/09/2021. URL: https://sites.unicef.org/about/execboard/files/09-15-accountability-ODS-English.pdf (cit. on p. 152).

Van der Aalst, W. M. (2013). “Business process management: a comprehensive survey”. In: *International Scholarly Research Notices* 2013 (cit. on pp. 8, 28).

Vincent, N. A. (2011). “Moral Responsibility”. In: vol. 27. Library of Ethics and Applied Philosophy. Springer. Chap. A Structured Taxonomy of Responsibility Concepts (cit. on pp. 6, 58).

Weske, M. (2007). *Business Process Management: Concepts, Languages, Architectures*. Springer (cit. on pp. 8, 27, 120).

Weyns, D. and Georgeff, M. (2009). “Self-adaptation using multiagent systems”. In: *IEEE Software* 27.1, pp. 86–91 (cit. on p. 34).

Weyns, D., Omicini, A., and Odell, J. (2007). “Environment as a first class abstraction in multiagent systems”. In: *Autonomous Agents and Multi-Agent Systems* 14.1, pp. 5–30 (cit. on p. 73).

White, S. A. (2004). “Introduction to BPMN”. In: *IBM Cooperation* 2.0 (cit. on p. 28).

Whittle, J., Sawyer, P., Bencomo, N., Cheng, B. H., and Bruel, J.-M. (2009). “RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems”. In: *2009 17th IEEE International Requirements Engineering Conference*. IEEE, pp. 79–88 (cit. on p. 32).

Wijngaarden, A. van (1966). *Recursive Definition of Syntax and Semantics: (proceedings Ifip Working Conference on Formal Language Description Languages, Vienna 1966, P 13-24)*. Stichting Mathematisch Centrum. Rekenafdeling (cit. on p. 19).

Woods, D. D. (2016). “The Risks of Autonomy: Doyle’s Catch”. In: *Journal of Cognitive Engineering and Decision Making* 10.2, pp. 131–133 (cit. on p. 3).

Wooldridge, M. (2009). *An introduction to multiagent systems*. John Wiley & Sons (cit. on pp. 2, 36).

Wooldridge, M. J. and Jennings, N. R. (1995). “Intelligent agents: Theory and practice”. In: *The Knowledge Engineering Review* 10.2, pp. 115–152 (cit. on p. 36).

World Health Organization (2015). *WHO Accountability Framework*. Accessed 15/09/2021. URL: https://www.who.int/about/who_reform/managerial/accountability-framework.pdf (cit. on p. 152).

Xu, J., Randell, B., Romanovsky, A., Rubira, C. M. F., Stroud, R. J., and Wu, Z. (1995). "Fault tolerance in concurrent object-oriented software through coordinated error recovery". In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. IEEE, pp. 499–508 (cit. on p. 26).

Xu, J., Romanovsky, A., and Randell, B. (1998). "Coordinated exception handling in distributed object systems: from model to system implementation". In: *Proceedings. 18th International Conference on Distributed Computing Systems*, pp. 12–21 (cit. on p. 26).

Xu, J., Romanovsky, A., and Randell, B. (2000). "Concurrent Exception Handling and Resolution in Distributed Object Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 11.10, pp. 1019–1032 (cit. on p. 26).

Yazdanpanah, V. and Dastani, M. (2016). "Distant Group Responsibility in Multi-agent Systems". In: *PRIMA 2016: Princiles and Practice of Multi-Agent Systems*. Vol. 9862. Lecture Notes in Computer Science. Springer, pp. 261–278 (cit. on p. 151).

Zahran, M. (2011). *Accountability Frameworks in the United Nations System*. Accessed 15/09/2021. URL: https://www.unjiu.org/sites/www.unjiu.org/files/jiu_document_files/products/en/reports-notes/JIU%20Products/JIU_REP_2011_5_English.pdf (cit. on p. 152).

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). "Developing multiagent systems: The Gaia methodology". In: *ACM Transactions on Software Engeering and Methodology* 12.3, pp. 317–370 (cit. on pp. 5, 57, 58).

List of Figures

List of Tables

List of Listings