Exceptions and Accountability for Robust Applications of JaCaMo

Matteo Baldoni[1 [0000−0002−9294−0408]] (✉),
Cristina Baroglio[1 [0000−0002−2070−0616]], Roberto Micalizio[1 [0000−0001−9336−0651]],
and Stefano Tedeschi[2 [0000−0002−9861−390X]]

[1] Università degli Studi di Torino – Dipartimento di Informatica, Torino, Italy
{matteo.baldoni,cristina.baroglio,roberto.micalizio}@unito.it
[2] Università della Valle d'Aosta - Université de la Vallée d'Aoste, Aosta, Italy
s.tedeschi@univda.it

Abstract. We present two extensions to the JaCaMo framework that support the realization of robust multi-agent organizations. Robustness amounts to the degree to which a system can function correctly in the presence of perturbations. The first extension provides an exception handling mechanism suited for MAS; the second one is grounded on the notion of accountability to create feedback chains among agents. Both extensions are built upon a uniform approach and provide high-level abstractions that facilitate the design and development of MAS that meet robustness requirements.

Keywords: JaCaMo · Engineering MAS · Exception Handling · Accountability.

1 Introduction

Robustness is defined as *"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions"* – generally called *perturbations* [16]. It is a crucial requirement of modern, distributed software systems [17,18,11,10]. Multi-Agent Systems (MAS) [22] are an effective approach to realize distributed systems by means of heterogeneous, and autonomous agents. Multi-agent *organizations* (MAOs), in particular, provide useful abstractions for modularizing code spread over many components, and orchestrate their execution by way of norms. JaCaMo [8] is one of the best-known platforms for implementing multi-agent organizations. However, it focuses on providing the means for capturing the normal, correct behavior of the system only. In particular, it lacks of structural mechanisms allowing agents to exchange and propagate information (feedback) when they face perturbations, thereby supporting robustness. As in [1], the availability of *feedback* about perturbations is crucial to build robust distributed systems. Also MAS robustness should ground on the ability to convey feedback about perturbations to the agents that can handle it. But since agents generally are peers, and are not related by relationships like caller-callee, as in programming languages, or parent-child, as in

the actor model, the realization of robustness should occur through the definition of appropriate distributions of responsibilities among the agents, that become part of the MAO.

This paper presents two extensions to the JaCaMo platform which allow building agent organizations that meet robustness requirements. The first borrows from software engineering the concepts of *exception* and *exception handling*, while the second relies on the notion of *accountability*. The two extensions differ in scope. Exception handling is suitable for treating perturbations anticipated at design time (i.e., exceptions) by activating handlers, that are also specified at design time. Accountability, instead, defines structured "channels" that agents impacted by perturbations can use at runtime to gain situational awareness about the situation of interest and then take actions. Raising and handling exceptions, as well as asking and returning for an account, will be tasks under the responsibility of specific agents. The two extensions provide the means for representing such tasks as goals, and for distributing the responsibilities of such goals to the capable agents seamlessly.

While in this paper we highlight the main features of the proposed extensions by means of a practical example, the details of the implementation can be found in [2,3,6,21] for exception handling and in [4,5,7] for accountability. The remainder of the paper is organized as follows. After a short introduction to JaCaMo in Section 2, Section 3 presents its extension with exception handling. Section 4, in turn, is focused on accountability. Section 5 briefly discusses the main aspects that characterize the integration of the two proposals into the JaCaMo implementation. Finally, Section 5 discusses a possible harmonization of the two extensions into a single proposal and sketches some usage scenarios.

2 Background: JaCaMo

JaCaMo [8] integrates three different programming dimensions: agents, environments and organizations. JaCaMo agents are programmed in Jason [9]. 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. The agents' environment consists of a dynamic and distributed set of shared *artifacts*, that are programmed in CArtAgO [20]. Each artifact provides to the agents the interface (set of operations), through which it can be used. Thus, agents can both perceive the artifact's observable state, reacting to events, and act upon an artifact by performing the artifact-provided operations.

JaCaMo organizations are programmed in Moise [15]. The organizational model structures the specification of an agent organization along three dimensions. 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

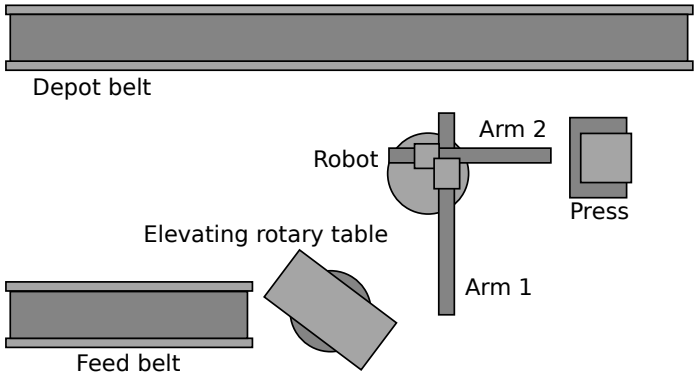

Fig. 1. An industrial production cell.

distributed to the agents. The *normative* dimension binds the previous two by specifying the role permissions and obligations for missions. The organizational infrastructure is designed as a part of the environment in which agents are situated by means of some dedicated organizational artifacts, upon which agents can perform operations.

At runtime, the organizational specification is translated into a *normative program*, written in a specific language, called NOPL [14]. The interpretation of such a program is performed by a dedicated interpreter, included in each organizational artifact, and regulates the functioning of the organization.

3 Exceptions in JaCaMo

To illustrate, we consider a production cell for metal plates inspired to [19] and reported in Figure 1. The system involves five robots (agents) that coordinate their activities for producing plates. The process can be realized as a JaCaMo organization where the organizational goal of producing a plate is decomposed into sub-goals the robots should achieve. Such a goal decomposition is reported in Figure 2, below. The figure also reports (in red) the roles to which each sub-goal is assigned.

We introduce the management of the possible malfunction of one of the motors of the elevating rotary table (*ERT*). Such a condition should be detected as soon as possible to stop the production and schedule repair. A first solution would be to add the treatment of the perturbation as a part of the original goal decomposition. This solution, however, is strongly discouraged by practice. In fact, mixing business logic and exception handling logic complicates the verification of processes as well as later modifications [13]. Our proposal is to keep the original goal decomposition distinct from any malfunction treatments, and introduce new abstractions for modeling treatments as exceptions to be raised

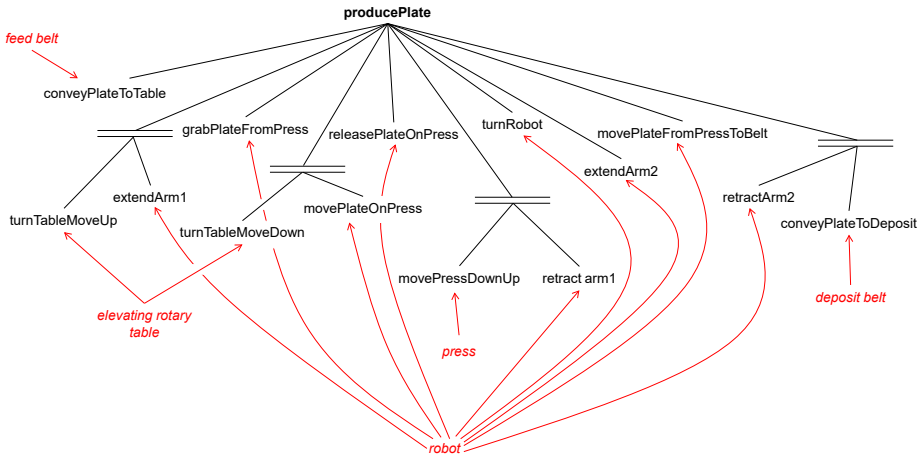

Fig. 2. Functional decomposition of the production cell scheme in JaCaMo.

and handled[3]. For instance, the following *Notification Policy* complements the goal decomposition.

```
<notification-policy id="npTable" target="turnTableMoveUp"
                     condition="scheme_id(S) & failed(S,turnTableMoveUp)">
    <exception-specification id="exMotor">
        <exception-argument id="motorNumber" arity="1" />
        <raising-goal id="notifyStoppedMotorNumber" />
        <handling-goal id="scheduleTableMotorFix" />
    </exception-specification>
</notification-policy>
```

The policy specifies how the ERT motor malfunction exception is handled by scheduling repair. In fact, the policy targets goal turnTableMoveUp, assigned to ERT in the original decomposition, and is activated whenever such a goal fails (see the condition expressed in NOPL syntax [14]). The policy then specifies the type of exception, exMotor, and its argument: the number identifying the motor affected by the problem (fundamental in order to act on the right motor). The exception, thus, amounts to a piece of information to be exchanged between some agent that detects the problem, and some that can handle it. To model this relationship, we extend the notion of goal, native in JaCaMo, in two ways. A **Raising Goal** is used to make an agent produce an exception (i.e., the structured piece of information), and raise it by making this exception available, through the organization, to the agents that can handle it. A **Handling Goal** is used to model the treatment of an exception when it becomes available. Both types of goals can be included in agent missions, as any other goal in JaCaMo.

[3] Source code available at http://di.unito.it/moiseexceptions.

This has an important consequence: whenever an agent enacts an organizational role, it is asked to commit to all the goals included in the missions associated with that role, including raising and handling goals. So, from the perspective of agent programming, the treatment of exceptions is completely transparent: agents just need to bring about their goals when asked to, independently of whether these goals are in the original decomposition or part of some notification policy.

In the example, the goals notifyStoppedMotorNumber and scheduleTableMotorFix must be included in the missions of agents having the right capabilities for completing them. So, either ERT itself could the exception raiser, or the exception could be raised by external, observing agent. An important feature of our proposal, in fact, is a clear separation of concerns among the agents where a perturbation occurs, where it is detected, and, then, where it is treated. That is, the agent whose goal fails may be different from the agent that actually detects the failure and raises the exception. And the agent handling the exception is usually different from the one that raised it.

4 Accountability in JaCaMo

Exceptions are suitable for treating perturbations that, anticipated at design time, can affect the achievement of some goals. Accountability allows agents to get runtime information to be used in their decision making. Specifically, a party, named account-taker (*a-taker*) is entitled to ask for an account about a goal of interest to another party, named account-giver (*a-giver*), that is obliged to provide such an account upon request. Through accountability, thus, agents have access to information otherwise inaccessible, and, hence, have greater awareness of what is going on in the overall system. This allows the agents to take advantage of opportunities, and to adapt to changing system conditions.

For instance, suppose the production cell is part of a production plant that should possibly never be stopped. Assume that an agent is in charge of supervising the production process. This agent can, under certain conditions, ask the *feed belt* robot the amount of plates still to be processed. Depending on such a number, the supervisor can decide to slowdown the production, in order to avoid a full stop. Also in this case, this behavior could be included within the original goal decomposition of the production process, but the result would be a mix-up between business and control logic. Our solution keeps separate business and control logic, and exploits accountability to allow the supervisor to obtain the needed information from the *feed belt* robot. The following *Accountability Agreement*, included in the definition of the organization, serves this purpose[4].

```
1 <accountability-agreement id="aa1" target="conveyPlateToTable" condition="true">
2     <account-template id="stock">
3         <account-argument id="availablePlates" arity="1" />
4         <requesting-goal id="requestRemainingStock" />
5         <accounting-goal id="notifyRemainingStock" />
```

[4] Source code available at http://di.unito.it/moiseaccountability.

```
6         <treatment-goal id="slowDownProduction"
7               when="account(stock,_,Args) & .member(availablePlates(N),Args) &
8                     N <= 10 & N > 0" />
9         <treatment-goal id="stopProduction"
10              when="account(stock,_,Args) & .member(availablePlates(0),Args)" />
11    </account-template>
12 </accountability-agreement>
```

The accountability agreement is the abstraction we offer to allow accounts to flow from a-givers to a-takers. Specifically, an accountability agreement targets a goal, e.g., conveyPlateToTable, which represents the object of the account. An agreement is activated by a requesting condition, that can even be true, meaning that it can be asked at any time throughout the execution. An important part of the agreement concerns the structure of the account, and how it can be asked and provided. The structure of the account is given as a list of arguments with their corresponding arity. In our example, a single argument availablePlates with arity one, is sufficient to convey the number of plates in the queue. Concretely, we leveraged the conceptual model presented in [4] and the formalization from [5,7].

To model the request and the notification of an account we extend JaCaMo goals, as we did for exceptions, by introducing the notions of **Requesting Goal** and **Accounting Goal**. Intuitively, when an agent wants to get some specific information outside its context, and has the permission to ask for them, the agent needs just to accomplish a requesting goal. This activates, by way of the normative system of the organization, the associated accounting goal specified in the agreement. For instance, when goal requestRemainingStock is marked as achieved, goal notifyRemainingStock is activated, and the agent responsible for it (i.e., *feed belt*) receives the obligation to carry it out. The agreement reports also an optional **Treatment Goal**. When specified, the goal specifies how the account should be addressed by the a-taker. In our scenario, there are two alternative treatment goals: one to be activated when the number of available plates is between 0 and 10, then the production is slowed down; and one to be activated when such a number is 0, and hence all the production cell is stopped.

Also in this case, the requesting, the accounting and the treatment goals are part of role missions, as for standard JaCaMo goals. By committing to such missions, agents take on the responsibility to perform these goals whenever a corresponding obligation is issued by the normative system of the organization.

5 Platform Integration

In order to include exception handling and accountability as primitive mechanisms in JaCaMo, and so allow the specification and the execution of MAOs with exceptions and accountability agreements, we had to work at multiple levels.

Specification level Provide the means to enrich the specification of an organization with a set of notification policies and accountability agreements;

Normative level Enable the enforcement of the normative behavior, yielded by notification policies and accountability agreements, by issuing obligations to achieve the related goals;

Infrastructural level Enrich the organizational infrastructure with the functionalities needed by agents to concretely raise or handle exceptions and provide or treat accounts.

In particular, we extended three components: (i) the organizational specification, (ii) the normative program, and (iii) the organizational artifacts. In JaCaMo, organizational specifications are written in XML. At runtime, the XML specification is translated into a set of NOPL normative programs. The organization management infrastructure is, then, realized through a set of artifacts, upon which the agents can operate. Such artifacts allow agents to interact with the organization, by perceiving its observable state and by executing operations. In order to capture exceptions and accountability, and allow the interpretation of our extended normative program, we enriched a specific class of artifacts, for scheme management. In the following, we briefly sketch the main features of our implementations.

Extending the Normative Program We introduced some new organizational facts which allow capturing the structure structure of notification policies and accountability agreements. For instance, notificationPolicy(NP,Target,Condition) is used to capture the fact that NP is a notification policy that concerns the goal Target and is activated when the specified condition holds (for instance, when the target goal fails). These facts, however, are not sufficient because we also need to capture dynamic facts that occur at runtime. These dynamic facts are produced as consequences of specific operations performed by agents on organizational artifacts. For example, the requesting condition for an accountability agreement may amount to the failure of some goal. Since the not-extended JaCaMo only allows agents to mark goals as achieved, we had to extend it in order to enable the setting of other goal states. In particolar, we introduced the new artifact operation goalFailed(G), by which an agent can signal the failure of goal G. As a consequence, the dynamic fact failed(S,G) is added in the normative state.

Achieving additional goals At runtime, the normative systems checks, for each exception specification (or account template) in an active notification policy (accountability agreement) (i.e., with Condition satisfied), the presence of applicable goals. That is, goals whose when clause holds. For accountability agreements this amounts to enabling the requesting goal, giving the a-taker the possibility to request the account. Such a request is performed by achieving the requesting goal. When this happens, the normative system issues an obligation to achieve the accounting goal to the involved agent(s). For exception handling, the raising goal is enabled as soon as the notification policy condition is verified. Although this mechanism is common to all goals, raising and accounting goals are special since they produce a piece of knowledge (feedback), that is, an exception/account that is compliant with the given specification, to which they belong. To this aim, we

introduced two new artifact operations: raiseException(E,Args) allows an agent to raise an exception E with a list of arguments Args. Arguments are a set of ground predicates having the structures specified by the exception specification. Similarly, operation giveAccount(A,Args) allows an agent to produce an account that follows a given template, upon request. Exception/account arguments are are also made available to the other agents as artifact observable properties.

When an exception is raised, the normative system looks for handling goals that are enabled, that is, whose When clause holds. The same happens for treatment goals when the corresponding account id given. A handling (treatment) goal is enabled if its When condition holds, and if the precondition goals are satisfied. We additionally require that an exception (account) has actually been raised (given), and that the corresponding raising (accounting) goal be satisfied. In this way, we ensure that the agent is able to take advantage of the information provided by way of the raising (accounting) goal. It is possible that many handling (treatment) goals are concurrently enabled for the same exception (account). Obligations are, then, issued for each enabled goal. The involved agents can leverage the information, encoded by the exception (account), to enact the most appropriate countermeasures. Under this perspective, we introduced an additional artifact operation goalReleased(G) that allows agents to release a (failed) organizational goal G. This allows resuming the process, aimed at the achievement of the organizational goal, which would, otherwise, remain stuck. By releasing the failed goal, the agent signals the organization that the perturbation has been handled (possibly some alternative goal has been accomplished), and hence the goal progression can be resumed.

Introducing new norms As a final step, we had to provide the normative program with the norms needed to issue obligations towards agents for the newly introduced goals and to ensure a set of properties that guarantee the correct functioning of the exception handling and accountability mechanisms. As an example, some of these norms ensure that only designated agents can raise or handle specific exceptions, or that an account can be requested only if the condition of the enclosing accountability agreement actually holds (i.e., the accountability agreement is active).

6 Discussion and Conclusions

We have proposed two extensions to the JaCaMo framework that explicitly introduce exceptions and accountability as primitive concepts in the design and development of MAS applications, with the aim of increasing system robustness while preserving, at the same time, autonomy of the components (agents). Agents joining an organization are required to explicitly take on the responsibilities for providing feedback about the context where perturbation are detected, and for handling these perturbations 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 normal behavior of the system and the one to be put in place in case of perturbations, uniformly.

The two extensions that we presented follow a uniform approach. Indeed, we are currently working in order to integrate both of them into a single proposal. Interestingly, by adopting a more general perspective, it is possible to leverage a unified framework to deal with various scenarios. We report here three of them, that capture a number of typical situations. These scenarios, together with a set of agent programming patterns designed to deal with such situations are discussed in detail in [7].

Firstly, it is worth noting that exception handling can be effectively conceived, more generally, in terms of accountability relationships among agents. Accountability supports the realization of exception handling mechanisms such that, whenever an exceptional condition occurs, an account concerning that condition is reported *by default* to the agent(s) responsible for handling it. Terminologically, the a-giver can be identified as the *exception raiser*, the a-taker as the *exception handler*, and the account amounts to the raised exception. 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.

Additionally, accountability supports *information gathering*. Suppose an agent needs a piece of information in order to take a decision. The information is not directly accessible to it; however, by way of an appropriate set of accountability agreements specified by the organization designer, the agent can rely on fellow agents in the organization in order to retrieve it. By exploiting accounts specified at design time via accountability, agents can get pieces of information that are outside their scope, and use them in their local decision-making process.

Finally, let us consider a scenario that we call *context-aware adaptation*. An agent is interested in an event, which has an impact on the achievement of the agent's goals (e.g., it may represent an occasion the agent could profit, or some perturbation that may negatively impact on the efficiency by which the goal can be achieved). The occurrence of such an event may induce the agent to consider to change its behavior in order to adapt to the situation. The decision on if/how to react to the event depends on the context in which the event occurs, but this information is outside the scope of the agent; however, the agent can take advantage of the accountabilities of other agents, where it acts as a-taker. It is worth noting that the adaptation could be exploited not only in case of perturbations, but also in order to take advantage of positive opportunities that may arise during the execution.

These three scenarios demonstrate the usefulness of accountability in a wide range of situations. In exception handling, in particular, the social structure realized by accountability, orthogonal to the functional decomposition of the organizational task, is exploited for conveying relevant feedback to the agent apt to handle it. Approaching exception handling in this way has many advantages. First of all, the solution relies on the abstractions of agent-based architectures (e.g., goals, beliefs, norms, etc.), and does not need any special structure dedicated to the management of exceptions. In addition, the overall system enjoys

low coupling and high cohesiveness, two desirable software engineering properties [12]. Low coupling is gained since agents dependence is limited to the exchange of an exception, specified by a well-defined account template within the organization. High cohesiveness, instead, is obtained by ascribing the tasks of raising and handling exceptions to the agents that have the right functionalities to accomplish them.

Future work will be devoted to consolidating the implementation of the two presented extensions into a single comprehensive proposal and to its integration into the JaCaMo distribution. We also aim at further validating benefits of the proposed approach in real-world applications. As hinted by the production cell example used in this paper, the domain of Industry 4.0 seems particularly promising.

Acknowledgments. This publication is part of the project NODES which has received funding from the MUR –M4C2 1.5 of PNRR with grant agreement no. ECS00000036.

References

1. Alderson, D.L., Doyle, J.C.: Contrasting views of complexity and their implications for network-centric infrastructures. IEEE Tr. on Sys., Man, and Cyber. **40**(4) (2010)
2. Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., Tedeschi, S.: Demonstrating Exception Handling in JaCaMo. In: Dignum, F., Corchado, J.M., De La Prieta, F. (eds.) 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. Lecture Notes in Computer Science, vol. 12946, pp. 341–345. Springer (2021). https://doi.org/10.1007/978-3-030-85739-4_28, https://doi.org/10.1007/978-3-030-85739-4_28
3. Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., Tedeschi, S.: Distributing Responsibilities for Exception Handling in JaCaMo. In: Endriss, U., Nowé, A., Dignum, F., Lomuscio, A. (eds.) Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems. pp. 1752–1754. AAMAS '21, International Foundation for Autonomous Agents and Multiagent Systems (2021), http://www.ifaamas.org/Proceedings/aamas2021/pdfs/p1752.pdf
4. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Reimagining Robust Distributed Systems through Accountable MAS. IEEE Internet Computing **25**(6) (2021). https://doi.org/10.1109/MIC.2021.3115450, https://doi.org/10.1109/MIC.2021.3115450
5. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Robustness Based on Accountability in Multiagent Organizations. In: Endriss, U., Nowé, A., Dignum, F., Lomuscio, A. (eds.) Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems. pp. 142–150. AAMAS '21, International Foundation for Autonomous Agents and Multiagent Systems (2021), http://www.ifaamas.org/Proceedings/aamas2021/pdfs/p142.pdf
6. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Exception handling as a social concern. IEEE Internet Computing **26**(6), 33–40 (2022). https://doi.org/10.1109/MIC.2022.3216272, https://doi.org/10.1109/MIC.2022.3216272

7. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Accountability in multi-agent organizations: from conceptual design to agent programming. Autonomous Agents and Multi-Agent Systems **37**(1), 1–37 (2023)
8. Boissier, O., Bordini, R.H., Hübner, J., Ricci, A.: Multi-agent oriented programming: programming multi-agent systems using JaCaMo. MIT Press (2020)
9. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons (2007)
10. Christie, S.H., Chopra, A.K., Singh, M.P.: Bungie: Improving fault tolerance via extensible application-level protocols. Computer **54**(5), 44–53 (2021). https://doi.org/10.1109/MC.2021.3052147
11. Christie, S.H., Chopra, A.K., Singh, M.P.: Mandrake: multiagent systems as a basis for programming fault-tolerant decentralized applications. Autonomous Agents and Multi-Agent Systems **36**(1) (2022). https://doi.org/10.1007/s10458-021-09540-8
12. Goodenough, J.B.: Exception handling design issues. SIGPLAN Not. **10**(7), 41–45 (Jul 1975)
13. Hagen, C., Alonso, G.: Exception handling in workflow management systems. IEEE Trans. Software Eng. **26**(10), 943–958 (2000). https://doi.org/10.1109/32.879818, https://doi.org/10.1109/32.879818
14. Hübner, J.F., Boissier, O., Bordini, R.H.: A normative organisation programming language for organisation management infrastructures. In: Coordination, Organizations, Institutions and Norms in Agent Systems V. Lecture Notes in Computer Science, vol. 6069, pp. 114–129. Springer (2009)
15. Hübner, J.F., Sichman, J.S., Boissier, O.: Developing organised multiagent systems using the MOISE+ model: Programming issues at the system and agent levels. Int. J. Agent-Oriented Softw. Eng. **1**(3/4), 370–395 (2007)
16. ISO/IEC/IEEE: Systems and software engineering - Vocabulary. 24765:2010(E) - ISO/IEC/IEEE International Standard (2010)
17. Jain, A.K., Aparico IV, M., Singh, M.P.: Agents for process coherence in virtual enterprises. Communications of the ACM **42**(3), 62–69 (1999)
18. Kalia, A.K., Singh, M.P.: Muon: designing multiagent communication protocols from interaction scenarios. Autonomous Agents and Multi-Agent Systems **29**(4), 621–657 (2015)
19. Lewerentz, C., Lindner, T.: Case study "production cell": A comparative study in formal specification and verification, pp. 388–416. Springer (1995)
20. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment Programming in CArtAgO, pp. 259–288. Springer US, Boston, MA (2009)
21. Tedeschi, S.: Exception Handling for Robust Multi-Agent Systems. Ph.D. thesis, Università degli Studi di Torino, Dipartimento di Informatica, Torino, Italy (2021)
22. Wooldridge, M.: An introduction to multiagent systems. John Wiley & Sons (2009)