The Knowledge Engineering Review (2025), **40**, e4, pp. 1–23
doi:10.1017/S0269888925000050

The Knowledge Engineering Review

CAMBRIDGE UNIVERSITY PRESS

RESEARCH ARTICLE

Exception handling in multi-agent oriented programming

Matteo Baldoni[1], Cristina Baroglio[1], Roberto Micalizio[1] and Stefano Tedeschi[2]

[1]Dipartimento di Informatica, Università degli Studi di Torino, Torino, Italy
[2]Università della Valle d'Aosta – Université de la Vallée d'Aoste, Aosta, Italy
Corresponding author: Matteo Baldoni; Email: matteo.baldoni@unito.it

Received: 21 December 2023; **Revised:** 10 April 2025; **Accepted:** 10 April 2025

Keywords: JaCaMo; Multi-agent Systems; exception handling; organization and institutiion; engineering multi-agent systems

Abstract

Exception handling has been successfully proposed in the past years as a simple yet powerful software engineering tool to promote modularity and decoupling, while also preserving robustness. Multi-agent systems (MAS) and organizations (MAOs), in turn, offer powerful abstractions to build distributed systems; current models and methodologies, however, fall short in addressing exception handling in a systematic way, not considering exceptions as part of their design. Thus, the problem is usually approached by *ad hoc* solutions that hamper code modularization and decoupling. In this work, we outline a vision of how exception handling in MAS can be granted by design. We present an extension of the organizational model and infrastructure adopted in JaCaMo, 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. In an organization, besides responsibilities for organizational goals, we propose to specify also responsibilities for managing exceptions, that is, for providing feedback about the context in which exceptions occur, and for handling it.

1. Introduction

A software program can encounter, during its executions, conditions that were not foreseen, sometimes by mistake, during the implementation. Think, for instance, to the common experience of a badly controlled for-loop that should initialize an array of values but exceeds its boundary. More interesting, however, is the case when the unforeseen situation is not due to a programming mistake, but rather to the weird behavior of a user (who, for instance, digits the string 'fiftyone' instead of the expected number 51), or to unforeseen environmental conditions, like a black-out. When this happens, the program becomes unreliable if unable to properly tackle the case.

Many programming languages allow programmers to tackle such cases by providing *exception handling* mechanisms, which can be used to enrich the code for specifying exceptions, that will be raised when, along the execution, certain conditions are met. An *exception* is often intended as an 'event that causes suspension of normal program execution' (ISO/IEC/IEEE, 2010). When an exception is raised suitable handlers are activated to manage the situation and allow the program to maintain a reliable behavior. Proposals like (Goodenough, 1975a,b,c) see exceptions as a means for extending the domain (i.e., the set of inputs for which effects are defined—as in the 'fifty-one' case), or the range (i.e., the effects obtained when certain inputs are processed—as in the array initialization case) of a piece of code, in a way that allows the *invoker* to tailor the management to the specific *context of use*—for instance, by writing a feedback to the user asking to insert digits instead of letters.

Cite this article: M. Baldoni, C. Baroglio, R. Micalizio, S. Tedeschi. Exception handling in multi-agent oriented programming. *The Knowledge Engineering Review* 40(e4): 1–23. https://doi.org/10.1017/S0269888925000050

© The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

In general, the full significance of an exception is known only *outside the detecting code*, which cannot determine unilaterally what is to be done after an exception is raised. Indeed, in many programming languages it is up to the 'invoker' to control the response to the exceptions that are raised by the 'invoked' method or function. For this reason, exceptions, besides notifying the occurrence of some unexpected event, are generally implemented as typed data structures, that allow passing to the handler all the relevant contextual information about the occurrence at issue (Hagen & Alonso, 2000).

Scaling from monolithic to distributed systems, the picture becomes more complex. Here, many threads/processes are executed in parallel. Exceptions might be raised in a thread/process that should be handled by another, but there is not a notion of 'invocation' to be leveraged upon. Some proposals concerning Akka and workflow management, for instance (Hewitt *et al*., 1973; Hagen & Alonso, 2000; Gupta, 2012; Goodwin, 2015), organize the execution of threads by way of parent–child relationships: when a child process raises an exception, its parent handles it. When we come to multi-agent systems (MAS) (Wooldridge, 2009), however, no structural relationships, like the mentioned invoker-callee and parent–child, exist. Thus, although it may happen that an agent failure has an impact on the activities carried out by other agents, even identifying which agents should be capable of raising which exceptions, and which other agents should be capable of handling them, is challenging. The reasons are that agents are peers which interact by message exchange, and each agent is autonomous in its deliberative process—even when it cooperates and coordinates with others in order to accomplish its goals. For instance, consider a doctor and one of his or her patients. The patient emails some symptoms to the doctor without receiving any answer (because the message was lost). In this case, only the patient agent can realize that something went wrong and hence can raise an exception about the lack of feedback, which can then be handled by the doctor. Instead, in case the doctor cannot prescribe what requested, it is the doctor agent that can raise the exception, and the patient is the one who should handle it.

In this paper, we outline a vision of how exception handling can be seamlessly integrated in MAS design and development. To face the inherent need of coordination among autonomous agents, the *organization* metaphor has been used for a long time in MAS. Since the initial proposals, like (Corkill & Lesser, 1983; Dignum, 2009), agent organizations encompass an explicit structure of roles and relations; responsibilities of tasks are distributed among the participating agents by role adoption. Thanks to norms (Esteva *et al*., 2001), the structures of distributed responsibilities among agents have been enriched with structures of *social expectations*, making roles the anchoring point of social expectations on the behavior of the agents who will play them in the organization. In particular, normative multi-agent organizations (MAO) (Dardenne *et al*., 1993; Bauer *et al*., 2001; Bresciani *et al*., 2004; Hübner *et al*., 2007; Boella *et al*., 2008; Boissier *et al*., 2013; Boissier *et al*., 2020) provide mechanisms to publish, enact, adapt, monitor and enforce normative behaviors. Thus, once decided to adopt a role, with the accompanying norms, in order to participate to the organization, agents assume the responsibility of the targeted tasks and are expected to accomplish their duties.

Under this perspective, we claim that the raising and handling of exceptions are *tasks* that should be encompassed in the MAS as part of the specification of its organization, consequently falling under the *responsibility* of specific agents. More specifically, the contribution of this paper is twofold. First, the paper introduces a model that defines exceptions at both the organizational and the agent level, explaining the key concepts. Second, building on the conceptual meta-model presented in Baldoni *et al*. (2022), we explain how the JaCaMo platform was modified in order to support the implementation of MAOs with exceptions, providing insights not only into the technical changes but also into the rationale behind them. Note that, although we focus on explaining how we grafted exception handling on JaCaMo's architecture, the approach is general and could be adopted beyond the community strictly working with JaCaMo. We help the explanation by illustrating the realization of complex examples taken from the literature, and in particular inspired by Christie *et al*. (2021), that show how exceptions can be of use in practical, real-world scenarios.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

The paper is structured as follows. Section 2 recalls the main concepts about JaCaMo and introduces the extension of JaCaMo's meta-model with exception handling. Section 3 presents in detail how we integrated the proposed exception handling mechanism into JaCaMo's organizational infrastructure. Section 4 discusses the most relevant related works concerning exception handling in MAS. Discussion and conclusions end the paper.

2. Exception handling in JaCaMo

JaCaMo (Boissier *et al*., 2020) is one of the best-known platforms for programming MAO and integrates three different dimensions: agents, environments, and organizations. JaCaMo agents are programmed in Jason (Bordini *et al*., 2007). An agent is an entity composed of a set of beliefs, representing the agent 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 agent environment consists of a dynamic and distributed set of shared *artifacts* that are programmed in CArtAgO (Ricci *et al*., 2009). Each artifact provides to the agents the interface (set of operations), through which it can be used. Thus, agents can both perceive the artifact observable state, reacting to events, and act upon an artifact by performing the artifact-provided operations.

2.1 Specification of a basic organization

JaCaMo organizations are programmed in Moise (Hübner *et al*., 2007). The organizational model structures the specification of an agent organization along three dimensions (Hübner *et al*., 2010). 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 subgoals and how these subgoals 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. 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 agent goals. Obligations to achieve organizational goals are issued according to the normative specification of the organization and the current state of the system. An obligation is fulfilled when the corresponding goal is achieved by the recipient agent before a given deadline. During the execution of a scheme, the involved goals can be in the state *waiting* (the goal cannot be pursued yet because it depends on the satisfaction of other goals), *enabled* (the goal can be pursued; its preconditions have been satisfied), or *achieved* (the agents were able to achieve the goal). 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 (Hübner *et al*., 2009, 2010, 2011). The interpretation of such a 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: (i) a set of normative facts, either translated from the specification or added dynamically during the execution, (ii) a set of inference rules, and (iii) a set of norms. The specification of normative facts and inference rules follows a syntax that is similar to the ones used in Jason and Prolog. In case of prohibition, norm activation results in the failure of the action which triggered the norm. Otherwise, the result is the emission of an obligation directed toward some agent and concerning a state of the world that the agent ought to bring about.

Let us, now, introduce the running example that we will use along the paper. It is inspired by Christie *et al*. (2021), an article concerning fault tolerance in MAS. With reference to Figure 1:

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

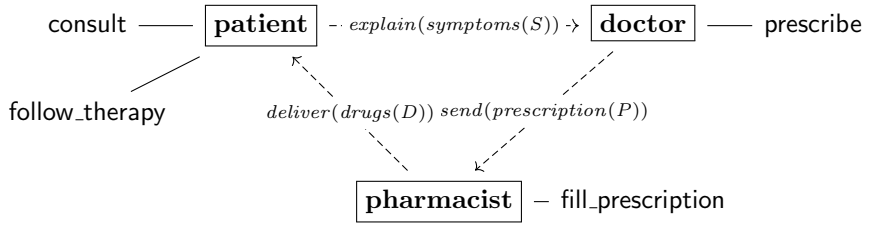

Figure 1. *The Therapy scenario*

Example 1 (Therapy) *The scenario involves a patient, a doctor, and a pharmacist:*

- *The* patient *consults the doctor explaining some symptoms and then waits for the medicines in order to follow the therapy.*
- *The* doctor *prescribes the medicines and communicates the prescription to the pharmacist.*
- *The* pharmacist *fills the prescription and has the prescribed drugs delivered to the patient.*

An additional constraint is that medicines must be taken within five days.

In order to implement the example in JaCaMo, it is necessary to realize an organization that involves agents playing the roles patient, doctor, and pharmacist. Such roles are part of the structural specification of the organization (Lines 3–7 in Listing 1). The functional specification of the organization (Lines 16–39) tells how the organizational goal (therapy) is structured into subgoals that are grouped into missions and assigned to roles by way of the normative specification (Lines 40–44). In our case, the subgoals are consult and follow_therapy (assigned to patient), prescribe (assigned to doctor), and fill_prescription (assigned to the pharmacist). In this context, *explain(symptoms)*, *send(prescription)*, and *deliver(drugs)* are actions executed while pursuing the goals associated with the roles (see Listings 3, 4 and 5). The agents playing the various roles will have plans that allow them to achieve their goals. In our example, in the plan for achieving the subgoal consult, the patient will explain the suffered symptoms to the doctor, while in the plan for achieving goal prescribe, the doctor will send the prescription to the pharmacist. Last but not least, for achieving goal fill_prescription, the pharmacist will deliver the drugs to the patient. Besides this, we require that follow_therapy be achieved within five days after the consult.

```
1  <organisational– specification>
2      <structural– specification>
3          <role– definitions>
4              <role id="patient"/>
5              <role id="doctor"/>
6              <role id="pharmacist"/>
7          </role– definitions>
8          <group– specification id="therapy_group">
9              <roles>
10                 <role id="patient" min="1" max="1"/>
11                 <role id="doctor" min="1" max="1"/>
12                 <role id="pharmacist" min="1" max="1"/>
13             </roles>
14         </group– specification>
15     </structural– specification>
16     <functional– specification>
17         <scheme id="therapy_sch">
18             <goal id="therapy">
19                 <plan operator="parallel">
20                     <goal id="cpf">
21                         <plan operator="sequence">
```

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

```
                        <goal id="consult" /> <goal id="prescribe" /> <goal id="fill_prescription" />
                    </plan>
                </goal>
                <goal id="follow_therapy" ttf="5 day" />
            </plan>
        </goal>
        <mission id="mPatient" min="1" max="1">
            <goal id="consult" />
            <goal id="follow_therapy" />
        </mission>
        <mission id="mDoctor" min="1" max="1">
            <goal id="prescribe" />
        </mission>
        <mission id="mPharmacist" min="1" max="1">
            <goal id="fill_prescription" />
        </mission>
    </scheme>
</functional-specification>
<normative-specification>
    <norm id="n1" type="obligation" role="patient" mission="mPatient" />
    <norm id="n2" type="obligation" role="doctor" mission="mDoctor" />
    <norm id="n3" type="obligation" role="pharmacist" mission="mPharmacist" />
</normative-specification>
</organisational-specification>
```

Listing 1. *Organization structural, functional, and normative specifications.*

As usual in JaCaMo, the execution of such plans is coordinated, within the organization, by the emission of obligations, whose activation depends on what occurred that far. By means of a built-in library (see Listing 2), obligations are mapped into agents internal goals that can, then, be practically pursued by the agent, for example, by executing some operations over the environment. In case of success, the corresponding organizational goals are set as achieved by means of the goalAchieved(...) primitive that is made available by the organizational infrastructure.

```
+obligation(Ag, Norm, What, Deadline)[artifact_id(ArtId), norm(_, Un)]
    :  .my_name(Ag) & (satisfied(Scheme, Goal) = What | done(Scheme, Goal, Ag) = What)
    <- .member(["M", Mission], Un);
       !fulfill_obligation(Scheme, Goal, ArtId, Mission).

+!fulfill_obligation(Scheme, Goal, ArtId, Mission)
    <- !Goal[scheme(Scheme)];
       goalAchieved(Goal)[artifact_id(ArtId)].
```

Listing 2. *JaCaMo library that maps organizational obligations onto agents' internal goals.*

Below, we report a skeleton implementation of a patient agent. The agent can handle two goals, consult (the doctor) and follow_therapy. The former is achieved by executing a plan that simply sends a message to the doctor, which explains the suffered symptoms. The latter involves waiting for medicine delivery, and then taking the prescribed drugs.

```
+!consult
    <- .send(doctor, tell, explain(symptoms([fever,headache]))).

+!follow_therapy : deliver(drugs(D))
    <- // Take the medicine D.

+!follow_therapy : not deliver(drugs(_))
    <- .wait({+deliver(drugs(_))});
       !follow_therapy.
```

Listing 3. *Excerpt of the* patient *agent's code.*

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

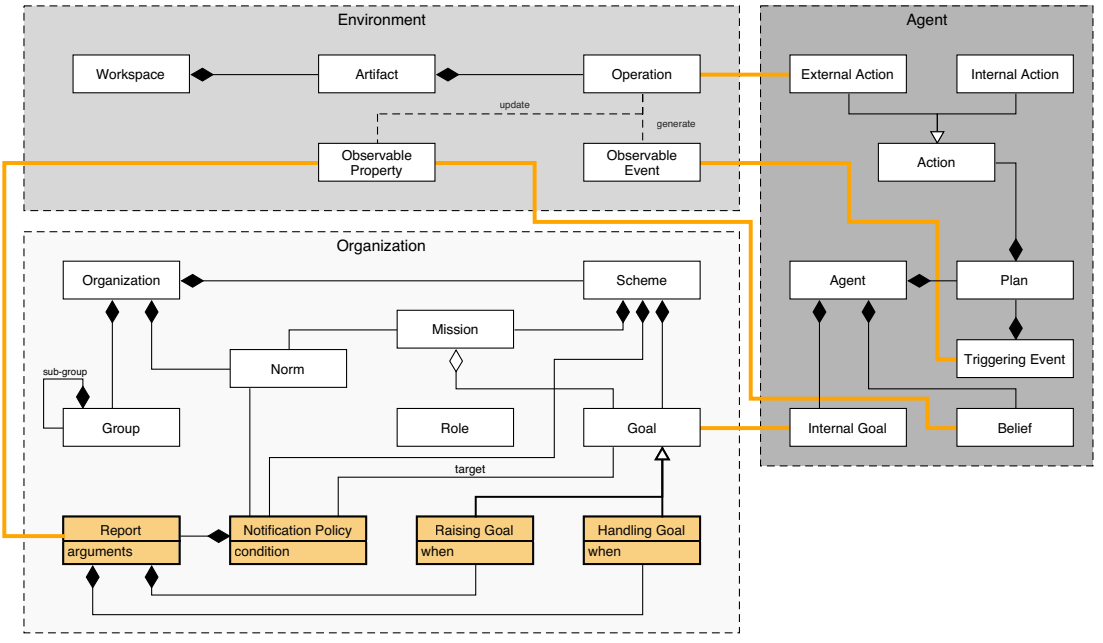

Figure 2. *JaCaMo's conceptual meta-model extended for exception handling. The newly added concepts are highlighted in orange. Orange lines also highlight the mapping among concepts belonging to different dimensions*

Similarly, the doctor and the pharmacist pursue their assigned goals as sketched hereafter.

```
1 +!prescribe : explain(symptoms(S))
2     <- !computePrescription(S,P);
3        .send(pharmacist, tell, send(prescription(P))).
4
5 +!computePrescription(S,P)
6     <- // analyze symptoms S, recover the patient's history ...
7        // write prescription P
```

Listing 4. *Excerpt of the* doctor *agent's code.*

```
1 +!fill_prescription : send(prescription(P))
2     <- // prepare parcel for drugs D in prescription P
3        .send(patient, tell, deliver(drugs(D))).
```

Listing 5. *Excerpt of the* pharmacist *agent's code.*

2.2 Adding exceptions

Let us, now, see how exception specification and exception handling are introduced in the picture. Figure 2 reports the meta-model of our proposal. It enriches a previous meta-model, described in Baldoni *et al*. (2022), with elements that were disregarded in that proposal, like the environment dimension and the mapping between concepts belonging to different dimensions, thus providing a complete picture. Such a mapping is central in the implementation of the infrastructure (for details, please, check Section 3).

In Baldoni *et al*. (2022), an exception is a piece of information that concerns the result of an operation, which is needed to interpret such a result and to identify the right prosecution. The functional specification of the organization includes, in the scheme, a set of *notification policies*, whose general template is reported in Listing 6. A notification policy concerns an organizational goal (`target`) and

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

Table 1. *Condition types for notification policies. When the organization is made of a single functional decomposition scheme instance* scheme_id(S) *can be omitted*

Type	Condition formula
always	true
goal failure	scheme_id(S) & failed(S, $target)
goal delay	scheme_id(S) & unfulfilled(obligation(_, _, done(S, $target, _), _))
goal achievement	scheme_id(S) & satisfied(S, $target)
custom	any valid NOPL formula

specifies an activation condition (`condition`): a logical formula that encodes the state of the organization, and the target goal in particular, in which the policy must be applied. Such conditions can amount to the failure, the delay, the achievement of a goal or some combination. Table 1 contains some relevant types of condition together with their corresponding NOPL formulas: clause scheme_id(S) identifies the functional decomposition scheme instance to which the condition refers. Placeholders starting with $ represent goal identifiers. Other conditions can be expressed as well, by directly specifying the corresponding NOPL formula as attribute. Note that it is possible to define many notification policies having the same goal as target, each having a different activation condition. In addition, a notification policy includes the definition of one or more exceptions that can possibly be raised, by means of the dedicated tag report. Each report specifies the structure of the information that will be exchanged, together with some raising and handling goals. Raising and handling goals are subject to further conditions that control their use.

```
1 <notification– policy id="ID" target="GOAL" condition="COND" type="exception">
2     [<report id="ExID">
3         [<argument id="ArgID" arity="N"/>] *
4         [<raise– goal id="GOAL" [when="COND"]?/> ] +
5         [<handle– goal id="GOAL" [when="COND"]?/> ] +
6     </report> ] +
7 </notification– policy>
```

Listing 6. *XML template for notification policies. + indicates one or more occurrences of the element between square brackets, ∗ indicates zero or more occurrences, and ? indicates optional elements.*

Let us now explain each field of a notification policy.

Reports capture the shape of the information that the agents responsible for raising the exception have to provide to those in charge for handling it (i.e., the feedback Hagen & Alonso, 2000; Alderson & Doyle, 2010). More in detail, reports include a set of arguments—predicates (with their arity) that must be instantiated and made available by the raising agents. Intuitively, a report is a *form* to be filled in by the agents in charge of raising the exception. The filled form amounts to the actual exception that is raised and made available to the handling agents. If a report includes multiple arguments, we assume that all of them will be instantiated. Each report also includes at least one *raising goal* and at least one *handling goal*. These are the organizational goals through which the exception is managed. Note that a single notification policy may include many reports. This feature is, for instance, handy when the system designer deals with events showing many facets, with possibly many consequences on the system. It will require several (generally uncoordinated) actions, by different agents, to properly tackle all of them at system level.

Raising goals and *handling goals*, denoted by tags raise − goal and handle − goal, are the organizational goals that allow managing exceptions. As standard goals in JaCaMo, raising and handling goals are included in missions, to which the agents, that play organizational roles, commit. Each report contains at least one raising goal and one handling goal, but it may encompass many of both types. So,

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

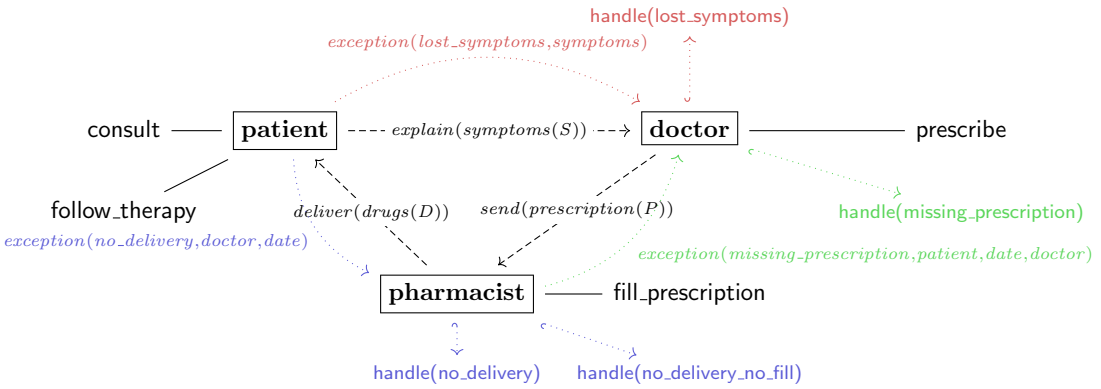

Figure 3. *The Therapy scenario extended with exception handling*

the same exception could be raised (and handled) in different ways in different circumstances, which are captured by the optional *when* clause, whose value corresponds to a state of the organization in which it makes sense to pursue that goal. Such a state amounts to a logical formula in NOPL, just like policy conditions (see Table 1).

Note that the act of raising an exception may require the joint efforts of many agents, as well as many agents may be involved in handling a raised exception. The involved agents may each be requested to carry out the same action, or they may be distributed different tasks which altogether produce the desired outcome. Thus, for each raising and handling goals, there may be many committed agents (exploiting JaCaMo's mission cardinality). Finally, raising and handling goals (like all goals in JaCaMo) might be structured, that is, they might be composed of many sub-goals, requiring the collaboration of many agents to collect and make the relevant information available.

The management of exception raising and handling through obligations and through the normative system of the MAO clearly identifies agents' responsibilities. In general, the agent that detects the conditions that will lead to raising an exception might not be equipped with the means to properly address the situation. It might not be able to determine the impact of the situation on the overall system. The exception handling mechanism we have introduced allows conveying relevant contextual information to those agents that are actually equipped with the means to promptly tackle them. This might be a complex process, involving many agents engaged in complex workflows. This is achieved by exploiting JaCaMo's standard notions of goal cardinality and decomposition.

Let us, now, illustrate the specification of notification policies with reference to the Therapy scenario and to Figure 3.

Example 2 (Exception Handling in Therapy) *Five days have passed and the patient has not received the needed medicines yet.*

The failure of the patient's goal follow_therapy may be due to different circumstances:

1. The doctor did not receive the symptoms sent by the patient;
2. The doctor received the symptoms, but did not produce the prescription;
3. The pharmacist filled the prescription, but the medicine did not arrive to the patient;
4. The pharmacist delayed the preparation of the medicine;
5. The pharmacist did not receive the prescription produced by the doctor.

These situations cause the failure of one or more organizational goals. We use these failures to discriminate between alternative exceptions and how to handle them. At the level of the organizational specification, we specified a notification policy (Listing 7) and added it to the scheme in Listing 1, with

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

target follow_therapy and with activation condition *unfulfilled(obligation(_, _, done(_, follow_therapy, _), _))*. In our scenario the deadline was missed, and indeed this condition corresponds to the condition type 'goal delay' in Table 1 (the scheme reference is omitted for brevity).

Cases (1) and (2) are detected by goal prescribe not being satisfied. In these cases, the patient is the one who must raise the exception, informing the doctor that the symptoms went lost: the patient will supply the date of the sending so that the doctor will sort out what happened (the doctor handles the exception). In Figure 3, this part is represented by the dotted red arrows. In the notification policy, the exception is specified with the name exception_lost_symptoms (see Lines 3–8 in Listing 7). The report includes two arguments, the date and the list of symptoms, the raising goal raise_exception_lost_symptoms that can be raised when condition *not satisfied(_, prescribe)* holds, and the handling goal handle_exception_lost_symptoms. The raising and handling goals are part of missions, see Lines 27–38 in Listing 7. Missions are associated with roles through the normative specification at Lines 40–44 in Listing 1.

Cases (3) and (4) become of interest when prescribe is satisfied, but the patient has not received the medicines. The patient should, in this cases, raise an exception toward the pharmacist (who will have the role of exception handler). This is possible because in JaCaMo a goal state is a public observable property of the organizational artifacts, so a patient can know whether a prescription was done by looking at the state of the organizational goal prescribe. The patient will supply the date of the consult and the doctor's name, so that the pharmacist can sort out what happened.

The pharmacis has two handling goals because it may either be the case that the prescription was filled (and the delivery failed), or the prescription for some reason has not been filled yet. The latter case may be due to the pharmacist, who is late and will then solve the problem. However, it may also be due to some error in the communication of the prescription from the doctor, which amounts to case (5). In this case, the pharmacist will, in turn, raise an additional exception toward the doctor, containing the patient's and doctor's names and the date of the consultation at issue, to solve the problem. This can be done by declaring the failure of the pharmacist's goal fill_prescription. In Figure 3, dotted blue arrows capture cases (3) and (4), while dotted green arrows capture case (5). In the notification policy, exception_no_delivery includes the date and the name of the doctor, as arguments. It includes the raising goal raise_exception_no_delivery that can be raised when the condition *satisfied(_, prescribe)* holds. Such a condition corresponds to the type 'goal achievement' in Table 1. The exception specification also includes two handling goals. The first, handle_exception_no_delivery, is used when the pharmacist has filled the prescription. The second, handle_exception_no_delivery_no_fill, is used when the pharmacist has not satisfied the goal fill_prescription, and will have to understand whether it is its fault or a lack of communication from the doctor. This last case calls for the specification of a further report, called exception_missing_prescription, which has two arguments: the patient's name, and the date of the consultation. It includes the raising goal raise_exception_missing_prescription that can be pursued when *satisfied(_, prescribe)* and *failed(_, fill_prescription)*. The handling goal, handle_exception_missing_prescription, is up to the doctor.

```
1  <notification-policy id="np1" target="follow_therapy"
2          condition="unfulfilled(obligation(_, _, done(_, follow_therapy, _), _))" type="exception">
3      <report id="exception_lost_symptoms">
4          <argument id="date" arity="1"/>
5          <argument id="symptoms" arity="1"/>
6          <raise-goal id="raise_exception_lost_symptoms" when="not satisfied(_, prescribe)"/>
7          <handle-goal id="handle_exception_lost_symptoms"/>
8      </report>
9      <report id="exception_no_delivery">
10         <argument id="date" arity="1"/>
11         <argument id="doctor_name" arity="1"/>
12         <raise-goal id="raise_exception_no_delivery" when="satisfied(_, prescribe) &amp;
13                 not raised(exception_lost_symptoms, _, _)"/>
```

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

```
14          <handle-goal id="handle_exception_no_delivery" when="satisfied(_, fill_prescription)"/>
15          <handle-goal id="handle_exception_no_delivery_no_fill" when="not satisfied(_, fill_prescription)"/>
16      </report>
17      <report id="exception_missing_prescription">
18          <argument id="date" arity="1"/>
19          <argument id="patient_name" arity="1"/>
20          <argument id="doctor_name" arity="1"/>
21          <raise-goal id="raise_exception_missing_prescription"
22              when="satisfied(_, prescribe) &amp;amp; failed(_, fill_prescription) &amp;amp;
23                  not raised(exception_lost_symptoms, _, _)"/>
24          <handle-goal id="handle_exception_missing_prescription"/>
25      </report>
26  </notification-policy>
27  <mission id="mPatient" min="1" max="1">
28      <goal id="consult"/> <goal id="follow_therapy"/>
29      <goal id="raise_exception_lost_symptoms"/> <goal id="raise_exception_no_delivery"/>
30  </mission>
31  <mission id="mDoctor" min="1" max="1">
32      <goal id="prescribe"/> <goal id="handle_exception_lost_symptoms"/>
33      <goal id="handle_exception_missing_prescription"/>
34  </mission>
35  <mission id="mPharmacist" min="1" max="1">
36      <goal id="fill_prescription"/> <goal id="handle_exception_no_delivery"/>
37      <goal id="handle_exception_no_delivery_no_fill"/> <goal id="raise_exception_missing_prescription"/>
38  </mission>
```

Listing 7. *Notification policy for the Therapy scenario.*

2.3 Agent programming with exceptions

Now coming to agent programming, Listing 8 shows the extended implementation of the *patient* agent, where the first two exceptions are raised.

```
1  +!consult
2      <- .send(doctor, tell, explain(symptoms([fever,headache]))).
3
4  +!follow_therapy : deliver(drugs(D))
5      <- // Take the medicine D.
6
7  +!follow_therapy : not deliver(drugs(_))
8      <- .wait({+deliver(drugs(_))});
9         !follow_therapy.
10
11 +!raise_exception_lost_symptoms
12     <- raiseException(exception_lost_symptoms,
13                       [date("23.03.2023"), symptoms([fever, headache])]).
14
15 +!raise_exception_no_delivery
16     <- raiseException(exception_no_delivery, [doctor_name(doctor), date("23.03.2023")]).
```

Listing 8. *Excerpt of the* patient *agent's code extended with exception handling.*

The first three plans allow the agent to pursue its assigned goals from the functional decomposition, while the last two are related to exception handling. More in detail, as soon as the scheme execution starts the agent receives two obligations: to pursue goals consult and follow_therapy. The obligations are automatically mapped onto the agent internal goals and the corresponding plans are activated. It is worth noting that the agent is able to successfully complete the plan at Line 7 only if the drugs are received from the pharmacist (see the plan context). Should this not happen before five days, the obligation to achieve goal follow_therapy would become unfulfilled. Consequently, the notification policy in Listing 7 would be triggered (see line 1). This causes the emission of new obligations concerning the raising/handling of exceptions.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

In our case, the patient is able to raise two exceptions (exception_lost_symptoms and exception_no_delivery), but it will actually raise only one of the two, depending on the context. The last two plans in the agent code serve this purpose: the agent raises the appropriate exception by means the primitive raiseException(...) (Lines 12 and 16), which takes in input a list of arguments that complies with the expected report specification. Such arguments are made available to the handler agents as observable properties of the organizational artifacts.

When raising exception_lost_symptoms, the agent will have to provide the consultation date and the symptoms, while when raising exception_no_delivery, the agent will have to provide the date and the doctor's name. The reason is that the first exception is to be handled by the doctor, while the second by the pharmacist. Since the pharmacist may receive prescriptions from multiple doctors, the information is needed to identify the right one. This example shows that report arguments encode relevant contextual information that must be provided from the raiser to the handler in order to allow the latter to cope with the exception at hand.

The achievement of a raising goal implies the emission of an obligation toward the agents that should achieve the associated handling goals. In our example, the raising of exception_lost_symptoms enables the handling goal handle_exception_lost_symptoms, which is in charge to the doctor. The exception exception_no_delivery, in turn, encompasses two different handling goals (in charge to the pharmacist). As explained the two goals are enabled in different circumstances, that are specified by the when condition.

Listing 9 below reports the code of the pharmacist agent, extended for handling the exception_no_delivery, raised by the patient.

```
1  +!fill_prescription : send(prescription(P))
2      <- // prepare parcel for drugs D in prescription P
3         .send(patient, tell, deliver(drugs(D))).
4
5  +!handle_exception_no_delivery : send(prescription(P))
6      <- // prepare parcel for drugs D in prescription P
7         .send(patient, tell, deliver(drugs(D))).
8
9  +!handle_exception_no_delivery_no_fill : send(prescription(P))
10     <- // prepare parcel for drugs D in prescription P
11        .send(patient, tell, deliver(drugs(D)));
12        goalAchieved(fill_prescription).
13
14 +!handle_exception_no_delivery_no_fill : not send(prescription(_))
15     <- goalFailed(fill_prescription).
16
17 +!raise_exception_missing_prescription
18     :  raised(exception_no_delivery, Args)[raiser(Patient)] &
19        .member(date(Date),Args) & .member(doctor_name(Doctor),Args)
20     <- raiseException(exception_missing_prescription,
21                       [date(Date), patient_name(Patient), doctor_name(Doctor)]).
22
23 +send(prescription(_)) : goalState(_, fill_prescription, _, _, failed)
24     <- resetGoal(fill_prescription).
```

Listing 9. *Excerpt of the* pharmacist *agent's code extended with exception handling.*

The plans at Lines 5, 9, and 14 target the handling goals. The first one is triggered when the agent, which has filled the prescription, receives the exception that was raised by the patient. This means that the drugs went lost before reaching the patient. The exception is handled by organizing another delivery of the prescribed drugs.

The remaining two plans capture cases (4) and (5) in Example 2. Here the pharmacist did not fill the prescription. This could be due to many reasons, we consider two alternatives: (1) the preparation for some reason was delayed and (2) the pharmacist did not receive the prescription from the doctor. The two plans in Listing 9 tackle the two cases. While in the first case the exception is handled totally and

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

directly by the pharmacist that will simply fill the delayed prescription, the second case is more complex. The agent cannot activate the plan to achieve the goal fill_prescription because the context at Line 1 does not hold. The agent, thus, marks the organizational goal as failed by means of the goalFailed(...) operation, at Line 15. This operation is provided by our the extended infrastructure and allows agents to proactively signal the failure of their organizational goals. The failure triggers the raising of a further exception (the last one specified in the notification policy). According to the policy, exception_missing_prescription must be raised by the pharmacist and handled by the doctor. The plan at Line 17 allows the agent to raise the exception. It is worth noting that the pharmacist may leverage the arguments provided beforehand by the patient while raising the previous exception (see the plan context). In this case, the date of the prescription and the patient's and doctor's names are forwarded to the doctor.

Finally, Listing 10 shows the doctor's code, extended with exception handling. The agent includes two plans that allow it to handle both the exceptions raised by the patient and by the pharmacist. To do so, the agent must achieve the corresponding handling goals. Here, we see again that the agent can leverage the information provided by the exception (i.e., the list of lost symptoms) to determine what has to be done.

```
1  +!prescribe : explain(symptoms(S))
2      <- !computePrescription(S,P);
3         .send(pharmacist, tell, send(prescription(P))).
4
5  +!computePrescription(S,P)
6      <- // analyze symptoms S, recover the patient's history ...
7         // write prescription P
8
9  +!handle_exception_lost_symptoms
10     :  raised(exception_lost_symptoms, Args)[raiser(Patient)] &
11        .member(date(D), Args) & .member(symptoms(S), Args)
12     <- !computePrescription(S,P);
13        .send(pharmacist, tell, send(prescription(P)));
14        goalAchieved(prescribe).
15
16 +!handle_exception_missing_prescription : explain(symptoms(S))
17     <- !computePrescription(S,P);
18        .send(pharmacist, tell, send(prescription(P))).
```

Listing 10. *Excerpt of the* doctor *agent's code extended with exception handling.*

3. Exceptions as part of the organization infrastructure

To include exception handling as a primitive mechanism in JaCaMo[1], and so allow the specification and the execution of MAOs with exceptions, we had to work at three different levels:

1. **Specification level**: we had to provide the means to enrich the specification of an organization with a set of notification policies;
2. **Normative level**: we had to enable the enforcement of the normative behavior, yielded by notification policies, by issuing obligations to achieve raising and handling goals;
3. **Infrastructural level**: we had to enrich the organizational infrastructure with the functionalities needed by agents to actually raise and handle exceptions.

Figure 4 illustrates the general picture of JaCaMo organizational component enriched with all the newly introduced elements (in orange the major changes). 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

[1]The full code of JaCaMo extended with exception handling, together with some examples, is available in the official Moise repository, in a dedicated branch: https://github.com/moise-lang/moise. We are currently working on integrating the extension into the official JaCaMo release.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

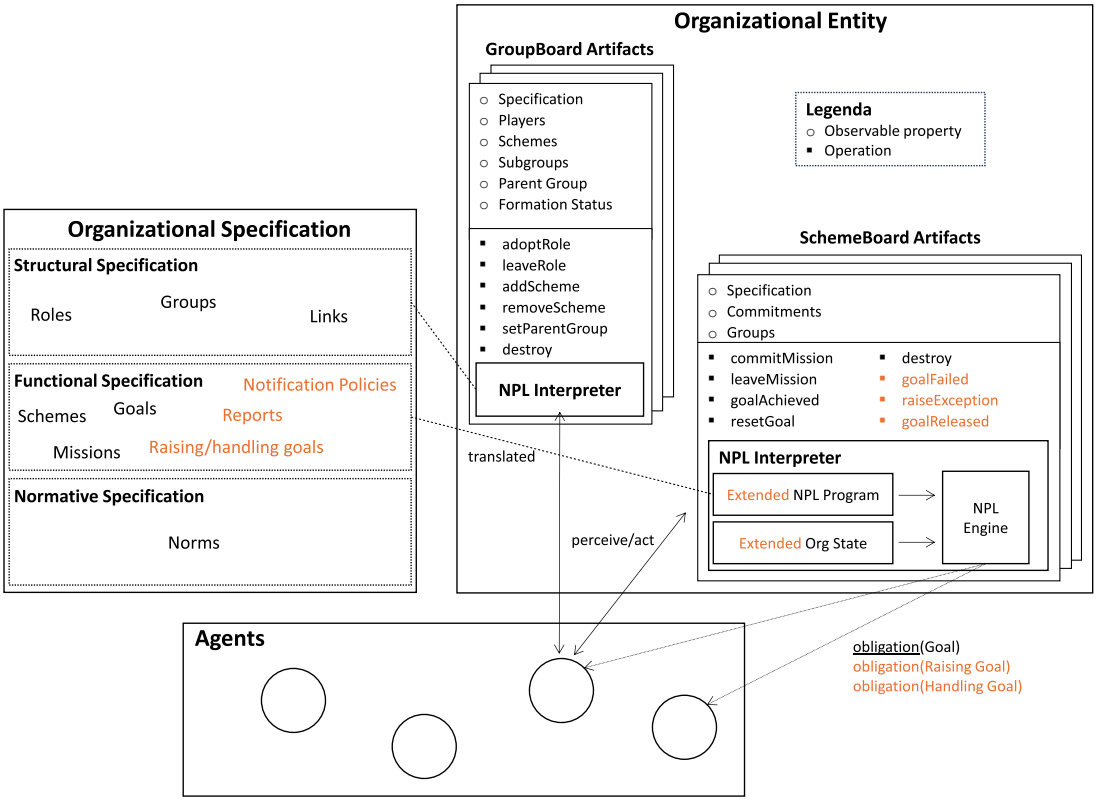

Figure 4. *General architecture of JaCaMo's organizational component. The parts affected by our extension are highlighted in orange*

translated into a set of NOPL normative programs, which address groups and schemes. 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 (JaCaMo's normative interpreter relies on these artifacts).

In order to capture exceptions and allow the interpretation of our extended normative program, we extended a specific class of artifacts for scheme management. Furthermore, we included a set of additional operations made available to the agents. The runtime functioning of our proposed extension is structured as follows.

3.1 Encoding notification policies in the normative program

Notification policies are specified in XML, but need to be translated into norms and rules to be interpreted by the normative system at runtime. To allow this to happen, we introduced some new organizational facts which allow capturing the notification policy structure. Table 2 lists them, together with an explanation of their meaning. So, 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 a consequence of specific operations performed by agents on some artifacts and regard the raising and handling goals.

For example, we just mentioned that a notification policy condition may amount to the failure of some goal, but standard JaCaMo only allows agents to mark goals as achieved, and hence we had

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

Table 2. *Normative facts capturing notification policies*

Fact	Meaning
notificationPolicy(NP, Target, Condition)	Notification policy with id NP targeting goal Target; Condition is the logical formula mapped from the policy attribute, as described in Table 1
report(E, NP)	Report for exception E defined within a notification policy NP
reportArgument(E, Functor, Arity)	Report argument—a ground first order predicate—with functor Functor and arity Arity associated with exception E. Multiple report arguments, each one with a different functor and arity may be defined for a given exception E
raisingGoal(RG, E, When)	Raising goal RG in the scope of a report for exception E. When is the logical formula mapped from the goal enabling condition. If no when attribute is specified, the condition is set to true
handlingGoal(HG, E, When)	Handling goal HG in the scope of report for exception E, with a When enabling condition (as for raising goals)

to extend it to widen the set of goal states. In particular, 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. Such a fact can appear in conditions triggering notification policies.

3.2 Enabling and accomplishing raising goals

At runtime, the normative systems checks, for each report in an active notification policy (i.e., with Condition satisfied), the presence of applicable (exception) raising goals. That is, raising goals whose when clause holds. This check is performed by means of the following rule.

```
1 enabled(S, RG) :-
2     raisingGoal(RG, E, When) &
3     notificationPolicy(NPol, _, Condition) &
4     report(E, NPol) &
5     Condition &
6     When &
7     goal(_, RG,  Dep, _, NP, _) & NP \== 0 &
8     ((Dep = dep(or,PRG)  & (any_satisfied(S, PRG) | all_released(S, PRG))) |
9         (Dep = dep(and, PRG) & all_satisfied_released(S, PRG))).
```

A *raising goal* RG is enabled when the Condition defined for the policy it belongs to, as well as its local When condition, hold, provided that its dependencies (preconditions) are satisfied. The predicate dep(...) is built-in NOPL and encodes the dependencies of a given goal, that is, the other goals that must be achieved (or released) before the goal can be pursued. Dependencies are deduced automatically from the functional decomposition.

Once a raising goal is enabled, the normative system issues an obligation to the involved agent(s). Although this mechanism is common to all goals, raising goals are special because they produce a piece of knowledge—an exception—that is compliant with the report that is given in the notification policy, to which the raising goal at issue belongs. To perform this, we introduced a new artifact operation: 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 facts reportArgument(E, Functor, Arity) for the report for exception E. The operation adds to the normative

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

state the dynamic fact raised(E, Ag, Args), where Ag is the name of the agent executing the operation and Args are the provided exception arguments. Note that the raised exception, together with its arguments is also made available to the other agents as an artifact observable property which has the shape raised(E, Args)[raiser(Ag)].

3.3 Enabling and accomplishing handling goals

Raised exceptions must be handled, so, when an exception is raised, the normative system looks for handling goals that are enabled. This is done by means of the following rule.

```
1 enabled(S, HG) :-
2     handlingGoal(HG, E, When) &
3     When &
4     raised(E, _, _) &
5     raisingGoal(RG,E,_) &
6     satisfied(S,RG) &
7     goal(_, HG,  Dep, _, NP, _) & NP \== 0 &
8     ((Dep = dep(or, PHG)  & (any_satisfied(S, PHG) | all_released(S, PHG))) |
9         (Dep = dep(and, PHG) & all_satisfied_released(S, PHG))).
```

Similarly to raising goals, a *handling goal* is enabled if its When condition holds, and if the precondition goals are satisfied. We additionally require that an exception has actually been raised, and that the corresponding raising goal has been satisfied. In this way, we ensure that the agent to which the handling goal is assigned, is able to take advantage of the information provided by way of the raising goal. It is possible that many handling goals are concurrently enabled for the same raised exception. Obligations are, then, issued for each enabled handling goal. The involved agents can leverage the information, encoded by the raised exception, to enact the most appropriate countermeasures and handle the exception.

In particular, the achievement of a handling goal could make no longer needed the goal whose failure triggered the exception. In such a case, the failed goal must be skipped. The new artifact operation goalReleased(G) allows agents to release an organizational goal G. It adds to the normative state the fact released(S, G). Executing this operation allows the appointed agents to notify the organization that the exception was handled. The execution of the scheme S can proceed because G is not of interest anymore. This allows resuming the process, aimed at the achievement of the organizational goal, which would, otherwise, remain stuck.

It is worth noting that releasing a failed goal is just a possibility. For instance, as an alternative the handling agent could decide to retry its achievement by resetting it, after the restoration of a consistent context. Of course, the choice for the best way to handle an exception is up to the agent that should handle it.

3.4 Norms for dealing with exceptions

As a final step, we had to provide the normative program with the norms needed to issue obligations toward agents for raising and handling goals and to ensure a set of properties that guarantee the correct functioning of the exception handling mechanism. To illustrate, some of these norms ensure that only designated agents can raise or handle specific exceptions, or that an exception can be raised only if the condition of the enclosing notification policy actually holds. We recall that a norm is composed of a name, a set of facts that represent the activation condition of the norm, and a consequence condition. The consequence of a norm activation can be either the emission of an obligation or a failure. In the latter case, this results in the failure of the action that triggered the norm.

The first addition concerns the achievement of raising and handling goals, which is to be coordinated through obligations by the normative system—as for any other organizational goal. In order to allow a

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

proper management, we had to modify the standard built-in norm for goal achievement, as reported in Listing 11: the norm does not issue anymore obligations concerning goals marked as failed or released (Lines 8–9).

```
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 11. *NOPL norm issuing obligations to achieve goals.*

Briefly, the norm says that if agent A is committed to mission M in scheme S and G is one of the mission goals (Line 3), such that G is either an achievement or a performance goal[2] (Lines 4–5), the scheme is well-formed (Line 6), and G is ready to be achieved, then an obligation directed toward A to achieve G must be emitted (Line 11). Note that in JaCaMo a scheme is well-formed when all the mission minimum and maximum cardinalities are respected, that is, when all the needed role-player agents have committed to their missions. Instead, a goal is considered ready for achievement as soon as it is enabled and it has not been already achieved, marked as failed or released, or its super-goal has been achieved (Lines 7–10).

We finally added some *regimented* norms to ensure consistency in the treatment of raising and handling goals. Regimented norms in Moise are those norms that result in a failure if the norm preconditions are not holding when the norm is triggered. Such norms ensure that agents cannot perform undesired actions. Specifically, the newly introduced norms are needed in order to ensure that agents only raise and handle exception that follow the specification. They can be divided into three groups: (i) norms that govern when exception can be raised, (ii) norms that govern what agents can raise exceptions, and (iii) what shape raised exceptions need to have.

The first group of norms deals with the fact that agents should be prevented from marking as "failed" goals that are not enabled, yet. Similarly, we also impose that exceptions can only be raised when an exceptional situation actually occurs, and not arbitrarily by the agents. The norm in Listing 12 regiments a failure if an agent tries to mark as failed an goal which is not enabled.

```
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 12. *NOPL norm regimenting goal failure.*

The norm in Listing 13, in turn, ensures that fact raised(E, Ag, Args) is asserted into the state of the organization only when the condition of the corresponding notification policy holds. It is worth noting that we allow the condition not to hold if the raising goal has been already achieved by some agent (Line 5).

[2]Note that Moise has the following goal types: achievement, performance, maintenance. Please, refer to Moise official documentation for further details.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

```
1 norm exc_condition_not_holding:
2     raised(E, Ag, Args) &
3     report(E, NP) &
4     notificationPolicy(NP, _, Condition) &
5     not (raisingGoal(RG, E, _) & (Condition | done(S, RG, Ag)))
6  -> fail(exc_condition_not_holding(S, E, Ag, Condition)).
```

Listing 13. *NOPL norm regimenting exception raising conditions.*

The rationale is that, after a successful handling of the exception the critical condition will likely stop holding. Nonetheless, the fact raised(E, Ag, Args), together with done(S, RG, Ag), keeps track of the fact that an exceptional situation occurred (and has been handled). done(S, G, Ag) is a built-in NOPL dynamic fact that is added to the normative program to denote that agent Ag has achieved goal G in scheme S.

The second group of norms ensure that only the designated agents can raise exceptions. The norm in Listing 14 inhibits the raising of exceptions by agents not committed to the mission including the corresponding raising goal.

```
1 norm exc_agent_not_allowed:
2     raised(E, Ag, Args) &
3     report(E, _) &
4     mission_goal(M, RG) &
5     raisingGoal(RG, E, _) &
6     not committed(Ag, M, S)
7  -> fail(exc_agent_not_allowed(S, E, Ag)).
```

Listing 14. *NOPL norm regulating agents allowed to raise exceptions.*

Finally, the last group of norms checks that exceptions, that are raised, are compliant with the specification. This check is not trivial and requires a number of rules. First of all, since raising goals must produce an exception (which is a piece of knowledge), they can be marked as "achieved" only when the corresponding exception has been provided by the raising agent.

The norm in Listing 15 ensures that a raising goal is marked as 'achieved' only when the corresponding raised(...) fact has been asserted. More in detail, a raising goal RG is not marked as achieved in those cases when the fact done(S, RG, Ag) is added to the normative state but the related exception has not been raised (see Line 5), because the goal achievement operation fails.

```
1 norm ach_rais_goal_exc_not_raised:
2     done(S, RG, Ag) &
3     raisingGoal(RG, E, _) &
4     not super_goal(SG, RG) &
5     not raised(E, _, _)
6  -> fail(ach_rais_goal_exc_not_raised(G, E, Ag)).
```

Listing 15. *NOPL norm regulating the achievement of raising goals.*

Note that raising goals may be composite, encompassing multiple subgoals, just like standard organizational goals. In this case, before achieving the root goal, all subgoals must be completed. When a raising goal is composite and require multiple subtasks, we require that the exception is actually raised before marking the root goal as achieved (see Listing 15, Line 4).

The norm in Listing 16 ensures that exceptions can be raised only by following a well-defined report specification.

Note that a fact of kind raised(...) can be asserted only when the right number of corresponding arguments has been asserted. Of course, each of them must comply with the expected format. This

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

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

Listing 16. *NOPL norm regimenting the raising of unknown exceptions.*

is verified by the norms in Listings 17–19. The one in Listing 17 ensures that the arguments of a raised(E, Ag, Args) predicate are ground predicates, i.e., they do not contain variables.

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

Listing 17. *NOPL norm regimenting exception arguments groundness.*

The norm in Listing 18 ensures that all the relevant information is provided, that is, that all the required arguments are instantiated.

```
1 norm exc_arg_missing:
2     raised(E, Ag, Args) &
3     report(E, _) &
4     reportArgument(E, ArgFunctor, ArgArity) &
5     not (.member(Arg, Args) & Arg = ..[ArgFunctor, T, A] & .length(T, ArgArity))
6  -> fail(exc_arg_missing(S, E, ArgFunctor, ArgArity)).
```

Listing 18. *NOPL norm regulating the absence of required exception arguments.*

Like in Prolog, the NOPL construct P = ..[F, T, A], used at Line 5, extracts the functor F, terms T, and eventually annotations A from 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 raised(E, Ag, Args). Indeed, this means that some of the information the agent was requested to provide is still missing.

The last norm, Listing 19 regiments the prohibition, for agents, to include undesired arguments when raising of an exception.

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

Listing 19. *NOPL norm prohibiting undesired exception arguments.*

The norm is triggered if an exception E is raised by some agent with a set of arguments Args and (at least) one of the arguments included in Args does not follow the specification for exception E (i.e., a corresponding reportArgument(...) fact is not present, see Lines 4–7). The result of all these norms is a failure in the exception raising action.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

4. Related work

Exception handling has been addressed in several papers of the MAS literature with different perspectives and purposes. Current proposals, however, do not capture the roles of the exception raiser and handler as primitive elements, and hence either they do not scale up or they violate the agents' autonomy.

Platon *et al*. (Platon, 2007; Platon *et al*., 2007b,a, 2008) look at exception handling as a tool that an individual agent can activate internally to preserve self-control despite the occurrence of exceptions. Jadescript (Petrosino *et al*., 2022, 2023) follows a similar approach. Here exceptions are special kinds of events that may occur internally to an agent and embody significant deviations from the nominal course of events (e.g., behavior failures). The language provides features to allow agents to detect and treat such kind of events. In contrast, our proposal considers exception handling as a social concern, addressed thanks to the cooperation of agents, which are assigned explicit (raising and handling) goals and can collaborate thanks to an unambiguous specification of exceptions.

A distributed perspective is taken by the *guardian* (Miller & Tripathi, 2004), a global entity that orchestrates the exception handling actions of a set of agents. When an exception is detected, the guardian applies a *recovery rule* that usually entails some exception handling procedures by the affected agents. In Klein and Dellarocas (1999), Dellarocas and Klein (2000), Klein and Dellarocas (2000), a decentralized evolution of the guardian model is proposed, where *sentinels*, equipped with handlers, may be plugged into the agents in case of an exception, in a coupled way. 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. Our proposal does not rely on special components such as sentinels or guardians, but seamlessly integrates exception handling into the agents themselves. However, the use of notification policies as specific constructs for specifying exceptions and their handling, allows a designer to keep separate the normal, expected behavior of an agent from its exceptional counterpart. This helps both the design and the maintenance of the agents themselves.

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 & 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*. An agent may be equipped with one or more *behaviors*, which map a collection of perceptions represented by *events* to a sequence of *actions*. Recently, the authors introduced a specific kind of event that represents any failure that an agent could handle, if interested (SARL.io, 2023). 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. One major limitation of the approach is that no explicit organization is set up for handling exceptions, In other words, no specific social structure is established among the agents, and hence, no mutual expectation about the handling of any failure event is possible (i.e., a failure event could pass uncaught legitimately). To overcome this issue, SARL spaces should be explicitly used to realize a social structure supporting exception handling as discussed in Baldoni *et al*. (2022, 2023b, 2025).

A drawback of the approach cited above is that no clear semantics is given to such a social structure. In Singh (2000), a social semantics for agent communication languages is presented. The work provides a foundational framework for understanding how agents exchange commitments and obligations during interactions, framing communication in terms of social relationships and norms. This aligns well with our approach to exception handling in MAS, where norms and obligations are central to defining how exceptions are raised and handled. Nonetheless, the approaches differs in scope. While we leverage the organization metaphor to provide the boundaries of the context where norms are defined, in Singh (2000) agents are peers that dynamically create commitments among each other as socially meaningful events that yield expectations on their behavior. *Social commitments*, indeed, are a valid conceptual tool to model responsibilities in MAS, also concerning the handling of exceptions. They are used, for

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

instance, in Jain *et al*. (1999), where agents face exceptional interactions by adjusting their commitments at runtime (i.e., by possibly canceling and creating new commitments). In Kalia and Singh (2015), Mallya and Singh (2005) exceptions are modeled via commitment-based protocols. Exceptions amount to commitments being violated. The flexibility of commitments allows creating preference structures over runs, and also merging specific handling protocols inside runs. 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 approach also deals with non-anticipated 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, however, does not scale-up well, as the authors state, splicing exception handlers at runtime requires a search through a library of handlers, that can be computationally demanding. On the other hand, inducing a preference structure over runs requires considerable design-time effort and extensive domain specific knowledge.

Similar considerations can be made for Mandrake (Christie V *et al*., 2022) and Bungie (Christie *et al*., 2021), two approaches to realize fault-tolerant agent interaction. They exploit several patterns for extending information-based protocols so as to optimize the way in which missing messages are detected and resent. One major difference between these approaches and ours lies in the fact that (Christie *et al*., 2021; Christie V *et al*., 2022) focus on failures affecting the exchange of messages among the agents, whereas our approach focuses on exceptions affecting the achievement of organizational goals. This has a substantial impact on the way in which an exception is conceptualized and handled. Recently, proposals have been made (Baldoni *et al*., 2025a,b; Chopra *et al*., 2025) for integrating information protocols with Jason. This line of research may provide a good starting point for integrating also exception.

Exception handling in interaction protocols has also been considered in Gutierrez-Garcia *et al*. (2009), where both interaction protocols and exception handlers are modeled through obligations. Exceptions are seen as abnormal situations in which agents cannot release an obligation. The obligation is canceled and, similarly to Mallya and Singh (2005), a handler is sought for in a repository. Exception handlers are modeled in terms of new obligations to be issued. This approach differs from ours for two main reasons: (1) it is not framed in an organizational dimension and (2) exceptions are not first-class objects specified since the design phase, but are just as abnormal situations emerging during the enactment of an interaction protocol.

The treatment of exceptions is of interest also in the perspective of self-adaptive systems. For instance, the Rainbow architecture (Garlan *et al*., 2009) aims at facing unexpected events and provides mechanisms to monitoring a target system and its environment, detecting events that denote opportunities for adaptation, selecting a course of actions 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. The approach bears some analogies with concept of notification policy 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.

5. Discussion and conclusion

In this paper we have presented an exception handling mechanism for MAOs that is seamlessly integrated with organizational concepts, such as responsibilities, goals and norms. This allows us to exploit the normative system of an organization not only for coordinating the expected, *correct*, behavior of the system, but also the exceptional behaviors. Robustness and correctness are complementary concepts: while correctness is *'the ability of software products to perform their exact tasks, as defined by their*

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

specification.' (Meyer, 1988), robustness guarantees that if exceptional cases do arise, the system will terminate its execution cleanly. We have shown that, by introducing a proper infrastructure, both properties can be supported by the normative system uniformly. This makes the treatment of exceptions, from their raising to their handling, an integral part of a MAO model, enabling a systematic and homogeneous treatment of exceptions, and simplifying the implementation of the agents, as we have exemplified in JaCaMo. We see exception handling in distributed systems as having an intrinsically *social nature*. From a practical point of view, the social aspect is captured by means of notification policies that specify, at the design of the organization, anticipated exceptions and how these have to be treated by cooperating agents. Notification policies are, thus, a software engineering tool, supporting the construction of robust, loosely coupled distributed systems, and with high cohesive components. Low coupling is achieved by limiting the interaction between the raising and handling agents to the exception itself, whose structure is part of the organization specification. The raising agent does not even need to know who will handle the exception. On the other hand, high cohesion is promoted because the designer has the flexibility to ascribe the goals of raising and handling exceptions to the agents that have the right functionalities to accomplish them. As we have shown, these tasks can even amount to complex goals, whose sub-goals are distributed among many agents.

The treatment of exceptions presented in this paper has some similarities with that of accountability introduced in Baldoni *et al*. (2021, 2023a). In both cases, we allow a designer to complement the functional decomposition with additional social structures (i.e., notification policies and accountability agreements, respectively). The two concepts, however, are substantially different in scope. Exceptions, borrowed from Software Engineering, are suitable for treating perturbations anticipated at design time by activating predetermined handlers. Accountability, instead, defines feedback 'channels' that agents can exploit at runtime to get information of interest falling outside their reach, and then take actions. In this second case, the decision about whether to request for an account rests on the agent that in an accountability agreement plays the role of accountability taker (*a-taker*). That is, the interaction through accountability is not triggered by the occurrence of an exception, but from an internal decision taken by an a-taker agent. As a future direction of research we are currently working on designing a unified framework that will allow dealing both with exceptions and accountability.

Acknowledgments. The work of Cristina Baroglio and Stefano Tedeschi is part of the project NODES which has received funding from the MUR—M4C2 1.5 of PNRR funded by the European Union—NextGenerationEU (Grant agreement no. ECS00000036).

References

Alderson, D. L. & Doyle, J. C. 2010. Contrasting views of complexity and their implications for network-centric infrastructures. *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans* **40**(4), 839–852.

Baldoni, M., Baroglio, C., Chiappino, G., Micalizio, R. & Tedeschi, S. 2022. Exception handling in SARL as a responsibility distribution. In *The 13th International Conference on Ambient Systems, Networks and Technologies (ANT)/The 5th International Conference on Emerging Data and Industry 4.0 (EDI40)*, Procedia Computer Science 201, 795–800. Elsevier. https://doi.org/10.1016/j.procs.2022.03.112

Baldoni, M., Baroglio, C., Galland, S., Micalizio, R., Outay, F. & Tedeschi, S. 2025. Interaction protocols in an imperative agent-oriented programming language: the case of BSPL and SARL. In *Proceedings of 24th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025*, IFAAMAS, Detroit, Michigan, USA.

Baldoni, M., Baroglio, C., Micalizio, R. & Tedeschi, S. 2021. Reimagining robust distributed systems through accountable MAS. *IEEE Internet Computing* **25**(6), 7–14. https://doi.org/10.1109/MIC.2021.3115450

Baldoni, M., Baroglio, C., Micalizio, R. & Tedeschi, S. 2022. Exception handling as a social concern. *IEEE Internet Computing* **26**(6), 33–40. https://doi.org/10.1109/MIC.2022.3216272

Baldoni, M., Baroglio, C., Micalizio, R. & Tedeschi, S. 2023a. Accountability in multi-agent organizations: from conceptual design to agent programming. *Autonomous Agents and Multi-Agent Systems* **37**(1). https://doi.org/10.1007/s10458-022-09590-6

Baldoni, M., Baroglio, C., Micalizio, R. & Tedeschi, S. 2023b. Towards exception handling in the SARL agent platform. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Cognitive Mimetics. The PAAMS Collection*, 403–408. Springer. https://doi.org/10.1007/978-3-031-37616-0_33

Baldoni, M., Christie V, S. H., Singh, M. P. & Chopra, A. K. 2025a. Orpheus: engineering multiagent systems via communicating agents. In *Proceedings Thirty-Nineth AAAI Conference on Artificial Intelligence, AAAI 2025*, Philadelphia, Pennsylvania, USA.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

Baldoni, M., Christie V, S. H., Singh, M. P. & Chopra, A. K. 2025b. Orpheus: programming protocol-based BDI agents. In *Proceedings of 24th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025, Demonstration Track*, IFAAMAS, Detroit, Michigan, USA.

Bauer, B., Müller, J. & Odell, J. 2001. Agent UML: a formalism for specifying multiagent software systems. *Software Engineering and Knowledge Engineering* **11**(3), 207–230.

Boella, G., van der Torre, L. & Verhagen, H. 2008. Introduction to the special issue on normative multiagent systems. *Autonomous Agents and Multi-Agent Systems* **17**(1), 1–10. https://doi.org/10.1007/s10458-008-9047-8

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A. & Santi, A. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* **78**(6), 747–761.

Boissier, O., Bordini, R. H., Hübner, J. & Ricci, A. 2020. *Multi-Agent Oriented Programming: Programming Multi-agent Systems Using JaCaMo*. MIT Press.

Bordini, R. H., Hübner, J. F. & Wooldridge, M. 2007. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. & Mylopoulos, J. 2004. Tropos: an agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems* **8**(3), 203–236. https://doi.org/10.1023/B:AGNT.0000018806.20944.ef

Chopra, A. K., Baldoni, M., Christie V, S. H. & Singh, M. P. 2025. Azorus: commitments over protocols for BDI agents. In *Proceedings of 24th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2025*, IFAAMAS, Detroit, Michigan, USA.

Christie V., S., Chopra, A. K. & Singh, M. P. 2021. Bungie: Improving fault tolerance via extensible application-level protocols. *Computer* **54**(5), 44–53. https://doi.org/10.1109/MC.2021.3052147

Christie V, S., Chopra, A. K. & Singh, M. P. 2022. Mandrake: multiagent systems as a basis for programming fault-tolerant decentralized applications. *Autonomous Agents and Multi-Agent Systems* **36**(1), 16. https://doi.org/10.1007/s10458-021-09540-8

Corkill, D. D. & 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)*, 748–756. William Kaufmann.

Dardenne, A., van Lamsweerde, A. & Fickas, S. 1993. Goal-directed requirements acquisition. *Science of Computer Programming* **20**(1), 3–50.

Dellarocas, C. & 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*, 95–102. IEEE.

Dignum, V. 2009. Handbook of research on multi-agent systems: Semantics and dynamics of organizational models.

Esteva, M., Rodríguez-Aguilar, J.-A., Sierra, C., Garcia, P. & Arcos, J. L. 2001. On the formal specification of electronic institutions. In *Agent Mediated Electronic Commerce: The European AgentLink Perspective*, 126–147. Springer Berlin Heidelberg.

Fischer, K., Schillo, M. & Siekmann, J. 2003. Holonic multiagent systems: a foundation for the organisation of multiagent systems. In *Holonic and Multi-Agent Systems for Manufacturing*, Lecture Notes in Computer Science **2744**, 71–80. Springer.

Garlan, D., Schmerl, B. & Cheng, S.-W. 2009. *Software Architecture-Based Self-Adaptation*, 31–55. Springer.

Gerber, C., Siekmann, J. & Vierke, G. 1999. Holonic multi-agent systems, Technical report, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH.

Goodenough, J. B. 1975a. Exception handling design issues. *SIGPLAN Notices* **10**(7), 41–45. https://doi.org/10.1145/987305.987313

Goodenough, J. B. 1975b. Exception handling: issues and a proposed notation. *Communications of the ACM* **18**(12), 683–696. https://doi.org/10.1145/361227.361230

Goodenough, J. B. 1975c. Structured exception handling. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'75, 204–224. ACM. https://doi.org/10.1145/512976.512997

Goodwin, J. 2015. *Learning Akka*. Packt Publishing Ltd.

Gupta, M. 2012. *Akka Essentials*. Packt Publishing Ltd.

Gutierrez-Garcia, J. O., Koning, J. & Ramos-Corchado, F. 2009. An obligation approach for exception handling in interaction protocols. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, **3**, 497–500. IEEE.

Hagen, C. & Alonso, G. 2000. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering* **26**(10), 943–958. https://doi.org/10.1109/32.879818

Hewitt, C., Bishop, P. & 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., 235–245.

Hübner, J. F., Boissier, O. & Bordini, R. H. 2009. A normative organisation programming language for organisation management infrastructures. In *Proceedings of the 5th International Conference on Coordination, Organizations, Institutions, and Norms in Agent Systems*, COIN'09, 114–129. Springer-Verlag.

Hübner, J. F., Boissier, O. & Bordini, R. H. 2010. From organisation specification to normative programming in multi-agent organisations. In *Computational Logic in Multi-Agent Systems*, 117–134. Springer.

Hübner, J. F., Boissier, O. & Bordini, R. H. 2011. A normative programming language for multi-agent organisations. *Annals of Mathematics and Artificial Intelligence* **62**(1), 27–53.

Hübner, J. F., Boissier, O., Kitio, R. & Ricci, A. 2010. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous Agents and Multi-Agent Systems* **20**(3), 369–400.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press

Hübner, J. F., Sichman, J. S. & Boissier, O. 2007. Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering* **1**(3/4), 370–395. https://doi.org/10.1504/IJAOSE.2007.016266

ISO/IEC/IEEE 2010. ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. ISO/IEC/IEEE 24765:2010(E), 1–418.

Jain, A. K., Aparico IV, M. & Singh, M. P. 1999. Agents for process coherence in virtual enterprises. *Communications of the ACM* **42**(3), 62–69.

Kalia, A. K. & Singh, M. P. 2015. Muon: designing multiagent communication protocols from interaction scenarios. *Autonomous Agents and Multi-Agent Systems* **29**(4), 621–657.

Klein, M. & Dellarocas, C. 1999. Exception handling in agent systems. In *Proceedings of the Third Annual Conference on Autonomous Agents*, AGENTS'99, 62–68, ACM.

Klein, M. & Dellarocas, C. 2000. A knowledge-based approach to handling exceptions in workflow systems. *Computer Supported Cooperative Work (CSCW)* **9**(3-4), 399–412.

Mallya, A. U. & Singh, M. P. 2005. Modeling exceptions via commitment protocols. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS'05, 122–129. ACM.

Meyer, B. 1988. *Object-Oriented Software Construction*, 2, Prentice Hall.

Miller, R. & Tripathi, A. 2004. The guardian model and primitives for exception handling in distributed systems. *IEEE Transactions on Software Engineering* **30**(12), 1008–1022.

Petrosino, G., Monica, S. & Bergenti, F. 2022. Robust software agents with the jadescript programming language. In *Proceedings of the 23rd Workshop "From Objects to Agents", Genova, Italy, September 1–3, 2022*, CEUR Workshop Proceedings 3261, 194–208. https://CEUR-WS.org. https://ceur-ws.org/Vol-3261/paper15.pdf

Petrosino, G., Monica, S. & Bergenti, F. 2023. Effective handling of exceptional situations in robust software agents. *Intelligenza Artificiale* **17**, 37–49. https://doi.org/10.3233/IA-230003

Platon, E. 2007. *Modeling Exception Management in Multi-agent Systems*. PhD thesis, Université Pierre et Marie Curie, France.

Platon, E., Sabouret, N. & Honiden, S. 2007a. Challenges for exception handling in multi-agent systems. In *Software Engineering for Multi-Agent Systems V*, 41–56. Springer Berlin Heidelberg.

Platon, E., Sabouret, N. & Honiden, S. 2007b. A definition of exceptions in agent-oriented computing. In *Engineering Societies in the Agents World VII*, 161–174. Springer Berlin Heidelberg.

Platon, E., Sabouret, N. & Honiden, S. 2008. An architecture for exception management in multiagent systems. *International Journal of Agent-Oriented Software Engineering* **2**(3), 267–289.

Ricci, A., Piunti, M., Viroli, M. & Omicini, A. 2009. *Environment Programming in CArtAgO*, 259–288. Springer US.

Rodriguez, S., Gaud, N. & 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)*, **3**, 103–110.

SARL.io 2023. Management of the failures and validation errors, sarl general-purpose agent-oriented programming language ("specification"), http://www.sarl.io/docs/official/reference/Failures.html. Accessed: 2024-09-16.

Schillo, M. & Fischer, K. 2002. Holonic multiagent systems. *Manufacturing Systems* **8**(13), 538–550.

Singh, M. P. 2000. A social semantics for agent communication languages. In *Issues in Agent Communication*, 31–45. Springer.

Wooldridge, M. 2009. *An Introduction to Multiagent Systems*. John Wiley & Sons.

https://doi.org/10.1017/S0269888925000050 Published online by Cambridge University Press