Exception Handling in Multiagent Organizations: Playing with JaCaMo

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

[1] Università degli Studi di Torino - Dipartimento di Informatica, Torino, Italy
firstname.lastname@unito.it

[2] Laboratoire Hubert Curien UMR CNRS 5516, Institut Henri Fayol, MINES Saint-Etienne, Saint-Etienne, France
Olivier.Boissier@emse.fr

Abstract. Agent organizations offer powerful abstractions to build distributed systems. Current models, however, lack of a systematic way to address exceptions as part of their design. Thus, exception handling is usually approached by ad-hoc solutions, that hamper code modularization and decoupling. 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.

Keywords: Exception Handling · Multi-Agent Organizations · Responsibility · JaCaMo

1 Introduction

Multiagent Systems (MAS) are widely used for the design and development of distributed, autonomous systems, characterized by multiple autonomous threads of execution that run in parallel, interact and coordinate with each other.

Several design methodologies and programming platforms that have been proposed in the field are grounded on the metaphor of *organization*. Key features of many organizational models (see e.g., [5, 6, 4, 7, 21]), are a functional decomposition of the organizational goal and a normative system. The functional decomposition defines how a complex goal can be decomposed into simpler sub-tasks, and allocates them to roles. By adopting roles in the organization, agents acquire responsibilities and duties they can discharge by performing tasks. Norms shape the scope of the responsibilities that agents take when joining the organization, capturing what they should do to contribute to the achievement of the organizational goal [9, 32, 33]. In particular, they coordinate the distributed execution by notifying the agents –by way of obligations– when tasks, that are in those agents' responsibilities by virtue of their participation to the organization, must be performed. Since agents are autonomous, Multiagent Organizations (MAO)

rely on sanctions to persuade them to achieve the organizational goals. Knowledge about possible sanctions, indeed, can be used in the deliberation process by which the agents decide which goals to pursue. An agent can decide to violate a norm (e.g., not discharge an obligation) when the sanction associated with the violation is balanced by the possible achievement of another goal of greater interest to the agent.

In this vision, norm violation is always the consequence of a deliberate decision by some agent. What is not considered is that the achievement of an organizational goal might be hindered by *exceptional* conditions, that are out of the agent's control. Sanctioning the agent, in this case, would not help the organization to get the job done.

What current MAO models lack is a systematic mechanism for treating exceptions as part of their design. Roughly speaking, *exception handling* amounts to equipping a system with the capabilities needed to tackle classes of abnormal situations, identified at design time. An *exception* is an "event that causes suspension of normal program execution" [1]. Therefore, the purpose of an exception handling mechanism is to provide the tools to (i) identify when an exception occurs, and (ii) apply suitable handlers, capable of treating the exception and recover. Thus, when an exception breaks the normal flow of execution, a predefined *exception handler* is executed to manage the specific situation. On its completion, the execution is possibly directed back to the normal flow of the program. *Raising* an exception is a way to signal that a given piece of the program cannot be performed normally; whereas, *handling* an exception refers to the set of instructions to be performed to restore the normal execution flow [10].

In this paper we show how an exception handling mechanism can be obtained within a MAO. Intuitively, besides the responsibilities about the tasks in the functional decomposition, we propose to specify also the tasks and responsibilities for managing exceptions, that is, for raising and handling them. Agents will take on these responsibilities as soon as they enact roles in the organization, as usual. We show how this mechanism can be grafted on the normative system of a MAO, and its advantages in terms of increased robustness in the execution. Specifically, we show how to introduce exception handling in the well-known JaCaMo multi-agent platform [5], integrating it both at a conceptual level, within the high-level abstractions that are provided by the model of JaCaMo's organizational component, and at a software level, by enriching its infrastructure.

2 Exception Handling is a Responsibility

The need of treating exceptions emerges from the desire of structuring and modularizing software, separating concerns into independent components that interact with each other. The seminal work by Goodenough on exceptions in programming languages [10–12], points out how exceptions allow the user of an operation to extend the operation domain (the set of inputs for which effects are defined), or its range (the effects obtained when certain inputs are processed). They allow the invoker tailoring an operation results or effects to the particular purpose

for which the operation is used, thus making them usable in a wider variety of contexts than would otherwise be the case. Consequently, an exception full significance is known only outside the detecting operation: the operation is not permitted to determine unilaterally what is to be done after an exception is raised. The invoker controls the response to the exception, that is to be activated. This increases the generality of an operation because the appropriate "fixup" will not be hard-coded inside the operation itself but, rather, it will vary from one use to the next, depending on the invoker's objectives. To make this possible the invoker should be provided with information about the failure.

Notably, the actor model [16] reflects this vision. Here, all computational entities are modeled as independent *actors*, communicating with others through message passing. In Akka[3], one of the most popular actor model frameworks [14, 13], actors are organized into a *supervision hierarchy*, which forms the basis of Akka's exception handling model. Specifically, actors are always created as children of some other existing actor, which supervises them and manages their lifecycle. The rationale is to break the task to perform down into sub-tasks to the point where each sub-task becomes simple enough to be performed by one single actor. Parent actors allow the composition of the sub-tasks, that are performed by their children, so as to meet the overall system objectives. Each time an actor faces a failure during the execution of a task, it can notify an exception to its parent actor, which, in turn, either implements suitable *supervision strategies*, or escalates the exception to its own parent. Paralleling Goodenough's vision, it is easy to see a parent actor as an operation invoker and a child actor as the invoked operation. This supervision technique can be conceived as a way to move the responsibility of handling an exception from the component that fails [13] to the one that, having delegated the sub-tasks, can determine the impact of a specific failure of one of them onto the concurrent execution of the others.

This perspective brings forward two important aspects of exception handling. *First*, it always involves two parties: a party that is *responsible for raising* an exception, and another party that is *responsible for handling* it. *Second*, it captures the need for some information/account from the former to the latter that allows coping with the exception.

Now, coming to MAS, we can observe that software structuring, modularization, and separation of concerns are brought to an extreme – agents are autonomous and less strictly coupled than in the actor model. Still the agents need to cooperate and they rely on one another to pursue their aims. So, it may well happen that an agent's failure in achieving some result has an impact on the tasks carried out by other agents. In this context, however, despite a few attempts [31, 24, 26, 15], exception handling – as postulated by Goodenough – has never been applied. Broadly speaking, what MAS currently lack is a clear distribution of responsibilities among agents for raising and handling exceptions. Indeed, a substantial difference between MAS and Actors is that agents are not structurally bound by parent-child relationships. Thus, when an agent meets a failure, it cannot easily determine which other agent could handle the related

[3] https://akka.io/.

exception. The agent that failed may ask the other agents but, due to autonomy, it would not be guaranteed that its request would ever be considered. The other agents, that are endowed with the right abilities, may prefer to achieve some other goal, and, in general, the system will not provide the means for persuading them to act otherwise. Multiagent Organizations (MAO) could provide the structure we need. Since MAOs, in essence, are built upon responsibilities, we claim that MAOs are naturally suited to encompass an exception handling mechanism.

In a MAO, each agent has only a partial view of the organizational goal, whose achievement is distributed among the agents. The normative system enables the orchestration of the activities by generating *obligations* according to some functional decomposition; obligations notify agents they should pursue certain goals. Of course, agents may fail, causing the suspension of the achievement of the organizational goal. When this happens, a normative system would typically issue a sanction to the agent that failed, because failure is ascribed to a misconduct of the agent. It is not possible for the agent to notify the impossibility to carry out its task due to external reasons. So, the limit of MAOs is that typically they are not accompanied by exception handling mechanisms that are oriented towards recovery.

We claim that the concept of responsibility not only allows modeling the duties of the agents in relation to the organizational goal, but that it also enables the realization of mechanisms for raising and handling exceptions that occur within the organization operation. When agents join an organization, they will be asked to take on also the responsibilities: 1) for providing accounts about the context where they detected exceptions, while pursuing organizational goals, and 2) if appointed, for handling such exceptions once the needed information is available. Responsibilities, thus, define the scope of the exceptions, expressed with respect to the organizational state, that agents ought to raise or handle.

In the following section we illustrate how to realize such a picture in JaCaMo.

3 Exceptions in JaCaMo

JaCaMo [5] is a conceptual model and programming platform that integrates agents, environments and organizations. Moise [22, 21, 20] implements the organization programming model. It comprises a *structural* dimension, specifying roles and groups, a *functional* dimension, including a set of schemes that captures how the organizational goals are decomposed into sub-goals, grouped into missions, and a *normative* dimension binding the previous two. Agents playing roles, in fact, are held to explicitly commit to missions, i.e., taking responsibility for mission goals.

3.1 Accommodating Exceptions in the Conceptual Model

Figure 1 shows the conceptual model of JaCaMo extended with some new concepts (in green) required to accommodate exceptions and their handling. We

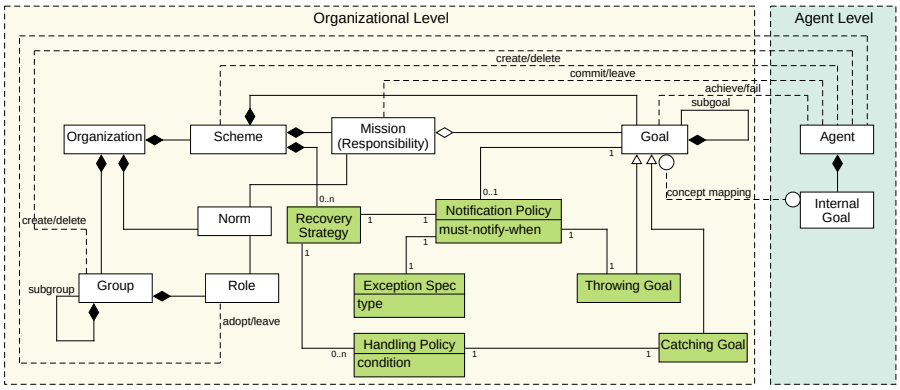

Fig. 1. Conceptual model of a JaCaMo organization extended for exception handling.

propose to enrich the schemes and missions of the functional specification of a JaCaMo organization with the following new concepts[4]:

Recovery Strategy encodes when and how a given exception is to be raised and handled within the organization. Its role is to relate the raising of an exception to the agent in charge of handling that exception. It includes a notification policy and one or more handling policies. A scheme may include several recovery strategies, each one addressing a specific exception that can possibly be raised during the execution of the scheme.

Notification Policy specifies when the exception must be raised. It is characterized by a condition (*must-notify-when*) denoting the state of the world corresponding to the exceptional situation that deserves to be signaled. A notification policy is also directly associated with a goal, representing the object of the exception (i.e., the goal that could not be completed), and with a throwing goal, that is enabled when *must-notify-when* holds.

Throwing Goal denotes the organizational goal of raising the exception.

Exception Spec encodes the kind of information to be produced by the agent raising the exception.

Handling Policy specifies a way in which the exception must be handled. It is characterized by a condition expressing the state of the world in which the policy is applicable and it is associated with a catching goal.

Catching Goal captures the course of action to follow for handling the exception and possibly remediate. The aim of its achievement is to restore the normal execution after an exception is raised.

Throwing goals and catching goals are special kinds of goal specification. They are incorporated into missions just like standard goal specifications are.

[4] The full code of Moise extended with exception handling, together with some examples, is available at http://di.unito.it/moiseexceptions.

As a result, missions provide also the means for distributing those responsibilities, that concern the management of exceptional situations. Policies, in turn, delimit the scope of such responsibilities, specifying when and how they are to be discharged.

3.2 Accommodating Exceptions in the Normative Program

The normative dimension of a JaCaMo organization is specified in NOPL (Normative Organization Programming Language) [17–19]. This language allows a programmer to specify the norms regulating the distribution of responsibilities among the roles. At runtime, a dedicated engine interprets the normative program and generates obligations, and possibly sanctions, depending on what the agents do in the organization.

A NOPL program is composed of: (i) a set of normative facts (that can change dynamically during the execution), (ii) a set of inference rules, and (iii) a set of norms. Norms have the form: $id : \phi \rightarrow \psi$; where id is a unique identifier, ϕ is a logical formula denoting the *activation condition* for the norm, and ψ is the *consequence* of the norm activation. ψ can either be a failure or the emission of an obligation directed towards an agent and concerning a state of the world the agent ought to bring about. To accommodate exceptions, we extended the normative program so as to properly handle the concepts introduced in Section 3. Specifically, we added the following facts.

`recoveryStrategy(RS)` denoting a recovery strategy with id `RS`.
`notificationPolicy(NP,Condition)` encoding a notification policy with id `NP`; `Condition` is a logical formula representing the *must-notify-when* condition.
`handlingPolicy(HP,Condition)` capturing that there is a handling policy with id `HP` which can be applied when the condition *Condition* holds.
`strategy_policy(RS,P)` encoding that a policy `P` belongs to a recovery strategy `RS`.
`policy_goal(P,G)` specifying the relation between a goal `G` and the policy `P`. Depending on the kind of policy (either a notification or a handling one), the goal will be a throwing goal or a catching goal.
`exceptionSpec(E)` encoding an exception specification with id `E`.
`policy_exceptionSpec(P,E)` denoting that the exception specification `E` is defined within the scope of a notification policy `P`.
`exceptionArgument(E,Arg)` denoting that the exception specification `E` encompasses an argument `Arg` – a first order predicate, possibly not ground.

While the previous facts reflect the extended conceptual model, the following ones are used to capture dynamic changes occurring during the execution.

`failed(S,G)` denotes a failure occurred while pursueing goal `G` in scheme `S`.
`released(S,G)` denotes that goal `G` has been released. The execution of the scheme `S` can proceed because `G` is not of interest anymore.

`thrown(S,E,Ag,Args)` denotes an exception thrown by agent `Ag`, compliant to the exception specification `E`; `Args` is a list of arguments, i.e., a set of ground predicates having the same structures of the arguments that are specified by `exceptionArgument(E,Arg)` for exception `E`.

All these new facts are used in rules and norms, that are specifically devised for raising and handling exceptions. In particular, rules allow defining when to enable throwing and catching goals, on the basis of the policy they belong to. For instance, for throwing goals we have defined the following rule.

```
enabled(S,TG) :- policy_goal(P,TG) &
    notificationPolicy(P,Condition) & Condition &
    goal(_, TG,  Dep, _, NP, _) & NP \== 0 &
    ((Dep = dep(or,PCG)  & (any_satisfied(S,PCG) |
    all_released(S,PCG))) |
    (Dep = dep(and,PCG) & all_satisfied_released(S,PCG))).
```

A throwing goal `TG` must be enabled as soon as the `Condition`, defined for the policy it belongs to, holds, provided that its dependencies (preconditions) are satisfied. `dep(...)` is a built-in NOPL predicate that encodes the dependencies of a given goal, i.e., the other goals that must be achieved before the goal can be pursued. Dependencies are obtained from the functional decomposition.

For catching goals an analogous rule applies.

```
enabled(S,CG) :- policy_goal(HP,CG) &
    handlingPolicy(HP,Condition) & Condition &
    recoveryStrategy(ST) & strategy_policy(ST,HP) &
    strategy_policy(ST,NPol) & policy_exceptionSpec(NPol,E) &
    thrown(S,E,_,_) & policy_goal(NPol,TG) &
    satisfied(S,TG) &
    goal(_, CG,  Dep, _, NP, _) & NP \== 0 &
    ((Dep = dep(or,PCG)  & (any_satisfied(S,PCG) |
    all_released(S,PCG))) |
    (Dep = dep(and,PCG) & all_satisfied_released(S,PCG))).
```

Similarly to throwing goals, a catching goal is enabled if the condition specified in the policy it belongs to holds, and if the precondition goals are satisfied. We additionally require that an exception has actually been raised, and that the corresponding throwing goal be satisfied. In this way, we ensure that the agent in charge of handling the exception (i.e., the one to whom the catching goal is assigned), is able to take advantage of the information provided by way of the throwing goal.

Notably, agents are asked to pursue throwing and catching goals by means of the standard built-in norms for goal achievement. We added some further regimented norms to ensure consistency. For instance, the following norm ensures that an exception can only be thrown if the condition of the corresponding notification policy holds, i.e., when an exceptional situation actually occurs.

```
norm exc_condition_not_holding:
    thrown(S,E,Ag,Args) & exceptionSpec(E) &
```

```
3     policy_exceptionSpec(NP,E) &
4     notificationPolicy(NP,Condition) &
5     policy_goal(NP,TG) & not (Condition | satisfied(S,TG))
6  -> fail(exc_condition_not_holding(S,E,Ag,Condition)).
```

At the same time, it is important to ensure that only the designated agents can throw exceptions. The norm inhibits the throwing of exceptions by agents that have not committed to the mission which encompasse the corresponding throwing goal.

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

3.3 Accommodating Exceptions in the Artifacts

In JaCaMo, the environment shared by the agents, from a same organization, is realized through a set of artifacts, upon which the agents can operate. We enriched a specific class of artifacts, for scheme management, to allow the interpretation of our extended normative program. Furthermore, we included three additional operations, available to the agents. The first operation, `goalFailed(G)`, allows agents to signal the occurrence of an exception, i.e., the failure of the fulfillment of one of the agent's responsibilities. Through such an operation, a fact `failed(S,G)` is added in the artifact state, enabling the triggering of new obligations for raising and handling the exception. Operation `throwException(E,Args)`, in turn, allows the agents in charge of raising an exception. Specifically, by means of this operation, an agent can provide information about the context in which the exception occured. Finally, `goalReleased(G)` allows the appointed agents to notify the organization that an exception was handled. 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, 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 a responsibility of the agent that should handle it.

4 Programming Agents with Exceptions: an Example

As already noted, in JaCaMo an agent enacting a role has to take on the responsibility of accomplishing certain goals by committing to given missions. This responsibility assumption can help agent programming. In principle, an agent program should allow discharging all of the agent's responsibilities; to this aim, an agent, playing a role, should be endowed with plans for satisfying each kind of obligation it could receive as role player.

The proposed exception handling mechanism follows this approach. The entire management of an exception, from its raising to its handling, is represented by means of goals, grouped into recovery strategies. When agents want to play roles in some organization, they are asked to take on responsibility not only for some missions, but also for some related recovery strategies. This makes the mechanism of exceptions completely transparent to the programmer, which will only need to know the responsibilities the agent is expected to discharge,

To illustrate, let us consider the *house building* example, originally introduced in [5], where an organization aims at building a house on a plot in a dynamic environment. Robustness, thus, becomes a critical feature of the organization, that has to coordinate multiple companies (i.e., agents) in charge of the various sub-goals, some of which can be executed in parallel, while others can only be executed in sequence. In such a construction scenario, exceptions are likely to occur: agents may fail to discharge their responsibilities for a wide number of reasons, and such failures could impact the organization as a whole.

4.1 Handling Goal Failure Exceptions

Let us consider goal `site_prepared`, which must be completed before any other step. Should the agent, which is in charge of it, face a failure, the whole house construction could not proceed. To make the organization robust against this eventuality, the programmer can extend the functional specification of the organization with the following recovery strategy.

```
<recovery-strategy id="rsSitePreparation">
    <notification-policy id="np1">
        <condition type="goal-failure">
            <condition-argument id="target"
                                value="site_prepared" />
        </condition>
        <exception-type id="site_preparation_exception">
            <exception-argument id="errorCode" arity="1" />
        </exception-type>
        <goal id="notify_site_preparation_problem" />
    </notification-policy>
    <handling-policy id="hp1">
        <condition type="always" />
        <goal id="handle_site_problem">
            <plan operator="parallel">
                <goal id="inspect_site" />
                <goal id="notify_affected_companies" />
            </plan>
        </goal>
    </handling-policy>
</recovery-strategy>
```

Listing 1. Recovery strategy targeting the failure of `site_prepared`.

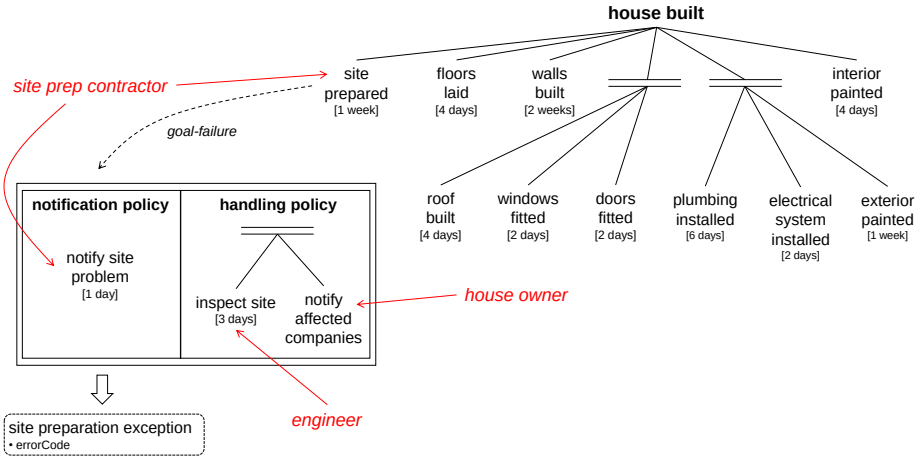

Fig. 2. Functional decomposition of the *house building* organizational scheme extended with the recovery strategy targeting the failure of `site_prepared`.

Notification policy `np1` specifies that, should a goal failure concerning `site_prepared` occur (see the condition at Lines 3-6), the throwing goal `notify_site_preparation_problem` should be enabled. Its purpose is to make the agent, that is responsible for it, provide the information that is needed for recovery. To this end, an exception type `site_preparation_exception` (Lines 7-9), specifying an `errorCode`, is defined. The handling policy `hp1`, in turn, encodes what needs to be done to solve the site preparation exception, after it has been raised and the error code was provided. In this case, the catching goal is composite (Lines 14-19): the site should be inspected and the other companies involved in the house construction should be notified. It is worth noting that the agents in charge of these goal will likely leverage the information provided. So, for instance, site inspection will be performed differently, depending on whether the error code denotes a flooding or the finding of archaeological remains.

Figure 2 illustrates the functional decomposition of the *house building* organizational scheme, extended with the recovery strategy presented above. Agents, that are responsible for some goals, are highlighted in red. Specifically, *house owner* has the responsibility for goal `notify_affected_companies`. An *engineer*, instead, is responsible for `inspect_site`: according to the raised exception, the result of the inspection, and its expertise, such an agent will deliberate the most appropriate countermeasures. Finally, a *site prep contractor*, in charge of `site_prepared`, is responsible for raising an exception by accomplishing goal `notify_site_preparation_problem`, when needed.

Having extended the organization specification with a recovery policy for `site_prepared`, we can now focus on how agents can be programmed. Listing 2 shows an excerpt of a possible implementation of the *site prep contractor* agent.

```
1 +obligation(Ag,_,done(_,site_prepared,Ag),_)
```

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

Listing 2. Code of the *site prep contractor* agent, raising the `site_preparation_exception`.

Notably, the agent discharges its responsibilities by way of two plans, reacting to two obligations. The first one (Line 1) refers to the achievement of goal `site_prepared`. The second one (Line 13) refers to the raising of an exception whenever goal `site_prepared` fails. In other words, the first obligation is issued in relation to a mission the agent is responsible for, whereas the second obligation is issued in relation to a recovery strategy, again under the responsibility of the agent. The obligation to achieve `site_prepared` is mapped onto an internal goal (Line 3). Should, for any reason, the agent fail to achieve such goal, the plan at Line 6 would be triggered.

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

Having discussed how an exception is raised, we now consider how it is handled. In our simple recovery strategy, the agent, that is responsible for handling

the failure of `site_prepared` is *engineer*. An excerpt of a possible implementation is shown in Listing 3.

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

Listing 3. Code of the *engineer* agent.

Also in this case, we focus on the plans the agent uses to discharge its responsibilities. The agent in Listing 3 is equipped with two alternative plans for discharging its responsibility, depending on the error code that comes with the `site_preparation_exception`. The choice is made possible because exception throwing allows specifying also some exception arguments, which are made available to the agents as observable properties of the organizational artifacts, and are automatically mapped on the agents' percepts. The first plan, triggered when the error code amounts to a flooding (Line 4), encompasses the execution of a site analysis (e.g., to estimate the damages) and then some fixes. In this case, goal `site_prepared` is released (Line 7), so that after the fix the construction can proceed. If, instead, the second plan is triggered, denoting the presence of archaeological remains, the course of actions to undertake is different. In this case the site is firstly delimited, then the remains are carefully removed, and finally `site_prepared` is reset, so that another attempt can be made in the site preparation.

4.2 Handling Goal Delay Exceptions

Let us now consider another kind of exception that may occur during the house building, i.e., a delay in the achievement of goal `windows_fitted` w.r.t. to the scheduled time of two weeks. We can easily specify a recovery strategy to deal with the exception, as illustrated in Listing 4.

```
1 <recovery-strategy id="rsWindowDelay">
2     <notification-policy id="np2">
3         <condition type="goal-ttf-expiration">
4             <condition-argument id="target"
5                                 value="windows_fitted" />
```

```
6           </condition>
7           <exception-spec id="windows_delay_exception">
8               <exception-argument id="weeksOfDelay" arity="1" />
9           </exception-spec>
10          <goal id="notify_windows_fitting_delay" />
11      </notification-policy>
12      <handling-policy id="hp2">
13          <condition type="custom">
14              <condition-argument
15                id="formula"
16                value="thrown(_,windows_delay_exception,_,Args)
17                       &amp; .member(weeksOfDelay(D),Args)
18                       &amp; D &gt;= 2"
19              />
20          </condition>
21          <goal id="handle_windows_fitting_delay" />
22      </handling-policy>
23  </recovery-strategy>
```

Listing 4. Recovery strategy targeting a delay in windows fitting in the house building scenario.

In this case, the Exception Spec of the notification policy amounts to `windows_delay_exception` with an argument `weeksOfDelay` (Lines 7-9). By this, the agent raising an exception about the delay of the `windows_fitted` goal, will also provide an estimation of the expected weeks of delay. This piece of information can, then, be used by the handling agent to reorganize the rest of his work. Indeed, according to the functional decomposition in Figure 2, goal `windows_ fitted` can be pursued in parallel with two other goals: `roof_built` and `doors_fitted`. While the latter is to be achieved in two weeks, the former takes up to four weeks. A designer can, then, define the recovery strategy in a way that the handling policy is applied only if the estimated delay exceeds two weeks (Line 18) [5]. The rationale is that, if the delay amounts to less than two weeks, the subsequent goals in the scheme are not affected, because they still depend on `roof_built`. In this particular case, even if the exception is raised, no corrective action is needed. On the other hand, if the delay exceeds 2 weeks, an obligation to achieve goal `handle_windows_fitting_delay` will be issued by the normative system, alerting the agent responsible for such a goal.

5 Exception Handling vs Message Passing

One might argue that robustness could be achieved in agent systems by relying on inter-agent messages. Message passing, however, brings on the system some substantial drawbacks. In first lieu, it strengthen agent coupling, as the following

[5] The condition is directly expressed as a NOPL formula in the condition argument. Since in JaCaMo the organizational specification is encoded in XML, some characters possibly occurring in NOPL formulas (such as &, >, and <) need to be escaped.

example highlights. Listings 5 and 6 show an implementation of the *site prep contractor* and *engineer* agents, respectively, where the exceptional situation due to `flooding` is handled through message passing.

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

Listing 5. Code of the *site prep contractor* agent, with exception handling realized through message passing.

The raising of an exception may be replaced with the sending of a message to notify the occurrence of a failure, and the corresponding error code. The point is that, since the responsibilities concerning the handling of such a situation are not clearly distributed among the agents, *site prep contractor* might not even know to whom such a message should be sent. Thus, in principle, such a notification should be broadcasted to all the agents. For the sake of simplicity, however, let's assume that *site prep contractor* knows that *engineer* and *house owner* might be willing to be notified about a failure in the preparation of the site. Thereby, in Listing 5, *site prep contractor* sends them the same notification message (lines 13-18). However, by doing this, we increase the coupling between the involved agents because the recipients of the message, as well as the shape of the message, are hard-coded inside the agent itself.

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

```
1 +exception(site_preparation_exception,Args)
2     : .member(errorCode(flooding),Args) &
```

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

Listing 6. Code of the *engineer* agent, with exception handling realized through message passing.

At the same time, agent development cannot follow a uniform approach to address the achievement of organizational goals and the handling of exceptions. Each exception must be addressed by defining *ad hoc* interaction protocols, whose specification falls outside the scope of the organization. To draw an analogy, this solution bears similarities with the definition of functions, in programming languages, where a particular return value denotes a failure (e.g., -1 is the typical failure value of Unix system calls). Of course, the implementation is possible, the drawback is that since the semantics of the values that are returned is twofold, the code will be burdened with checks (i.e., `if` statements) on the return value of every critical function to determine whether the function did its job or not. The same happens for agents, see for instance the code of *engineer* snipped in Listing 6. These two plans are specifically devised to capture the message from *site prep contractor*, and are not programmed by following the responsibilities of the agent (i.e., the set of obligations it has to fulfill), but hard coded as "`if`". Our mechanism, instead, allows to program agents just by looking at their responsibilities, capturing in a homogeneous way both the normal and exceptional behavior.

6 Conclusions and Related Work

Exception handling has been addressed by only a few papers in the MAS literature. Differently than the approach by Platon et al. [28, 30, 29, 31], where exception handling is seen as a tool that the individual agent can activate internally, to preserve self-control despite the occurrence of exceptions, the proposal we have made leverages on the distributed nature of exception handling, typical of programming languages and of the actor model [16], and suited to distributed systems made of cooperative parties, like MAO.

In [24, 8, 25], an approach based on a shared exception handling service is proposed. The service provides *sentinels*, that are equipped with handlers (inspired by research on management), to be plugged into existing agent systems. The

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

Mallya and Singh [26] propose to model exceptions via commitment-based protocols. Anticipated exceptions, occurring during the execution of an interaction protocol (i.e., deviations from the normal flow that occur often enough and are part of the model), are dealt with by specifying a hierarchy of preferred runs. Preferences can, then, be used to define exceptional runs. Exception handlers are treated as runs just like protocols. Handlers can be spliced inside a given protocol when an exceptional run is detected. The paper proposes also an approach to deal with unexpected exceptions (i.e., exceptions that are not part of the process model). Exception handlers, in this case, are constructed dynamically from a basic set of protocols. This approach seems promising, although some concerns related to scalability can be identified. Indeed, as the authors state, splicing exception handlers at runtime requires a search through a library of handlers, that can be computationally demanding. Conversely, inducing a preference structure over runs requires considerable design-time effort and extensive domain specific knowledge.

In the context of normative multi-agent systems, [15] propose an approach for exception handling in interaction protocols, where both interaction protocols and exception handlers are modeled through obligations in deontic logic. Exceptions are seen as abnormal situations in which agents cannot release an obligation. The obligation is canceled and, similarly to [26], a handler is sought for in a repository. Exception handlers are modeled in terms of new obligations to be issued. Despite both exploit obligations, this approach differs from ours because: (1) it is not framed in an organizational dimension; (2) exceptions are not first-class objects, constituting an account for an exceptional situation, but are rather simply conceived as abnormal situations emerging during the enactment of an interaction protocol. At the same time, exception handling is not conceived in terms of responsibilities taken by the agents. Finally, the proposal is mainly theoretical, no integration in any MAS platform is discussed.

To conclude, in this paper we presented an exception handling mechanism for MAOs that relies on the notion of responsibility. 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. This is a pretty novel use of normative systems, which are traditionally used to support the realization of *correct* systems. Robustness and correctness are

complementary concepts: while correctness is *"the ability of software products to perform their exact tasks, as defined by their specification."* [27], robustness guarantees that if different cases do arise, the system will terminate its execution cleanly. We have shown that, introducing a proper infrastructure, both properties can be supported by the normative system uniformly. The proposal could find application also in the area of Business Processes where tools like WS-BPEL [23] are currently used. An inspiration is provided by works like [2, 3].

Acknowledgments

Stefano Tedeschi's research project has been carried out thanks to the grant "Bando Talenti della Società Civile" promoted by Fondazione CRT with Fondazione Giovanni Goria.

References

1. ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. ISO/IEC/IEEE 24765:2010(E) pp. 1–418 (Dec 2010). https://doi.org/10.1109/IEEESTD.2010.5733835
2. Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., Tedeschi, S.: Accountability and responsibility in multiagent organizations for engineering business processes. In: Dennis, L.A., Bordini, R.H., Lespérance, Y. (eds.) Engineering Multi-Agent Systems - 7th International Workshop, EMAS 2019, Montreal, QC, Canada, May 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12058, pp. 3–24. Springer (2019)
3. Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R., Tedeschi, S.: Engineering business processes through accountability and agents. In: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, QC, Canada, May 13-17, 2019. pp. 1796–1798. International Foundation for Autonomous Agents and Multiagent Systems (2019)
4. Bauer, B., Müller, J., Odell, J.: Agent UML: A formalism for specifying multiagent software systems. Software Engineering and Knowledge Engineering **11**(3), 207–230 (2001)
5. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Sci. Comput. Program. **78**(6), 747–761 (2013). https://doi.org/10.1016/j.scico.2011.10.004
6. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems **8**(3), 203–236 (2004). https://doi.org/10.1023/B:AGNT.0000018806.20944.ef, http://dx.doi.org/10.1023/B:AGNT.0000018806.20944.ef
7. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming **20**(1), 3 – 50 (1993). https://doi.org/https://doi.org/10.1016/0167-6423(93)90021-G
8. Dellarocas, C., Klein, M.: An experimental evaluation of domain-independent fault handling services in open multi-agent systems. In: Proceedings Fourth International Conference on MultiAgent Systems. pp. 95–102. IEEE (2000)

9. Feltus, C.: Aligning Access Rights to Governance Needs with the Responsability MetaModel (ReMMo) in the Frame of Enterprise Architecture. Ph.D. thesis, University of Namur, Belgium (2014)
10. Goodenough, J.B.: Exception handling design issues. SIGPLAN Not. **10**(7), 41–45 (Jul 1975). https://doi.org/10.1145/987305.987313, https://doi.org/10.1145/987305.987313
11. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Commun. ACM **18**(12), 683–696 (Dec 1975). https://doi.org/10.1145/361227.361230, https://doi.org/10.1145/361227.361230
12. Goodenough, J.B.: Structured exception handling. In: Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 204–224. POPL '75, Association for Computing Machinery, New York, NY, USA (1975). https://doi.org/10.1145/512976.512997, https://doi.org/10.1145/512976.512997
13. Goodwin, J.: Learning Akka. Packt Publishing Ltd (2015)
14. Gupta, M.: Akka essentials. Packt Publishing Ltd (2012)
15. Gutierrez-Garcia, J.O., Koning, J., Ramos-Corchado, F.: An obligation approach for exception handling in interaction protocols. In: 2009 IEEE/WIC/ACM Int. J. Conf. on Web Intelligence and Intelligent Agent Tech. vol. 3, pp. 497–500 (2009)
16. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence. p. 235–245. IJCAI'73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1973)
17. Hübner, J.F., Boissier, O., Bordini, R.H.: 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. p. 114–129. COIN'09, Springer-Verlag, Berlin, Heidelberg (2009)
18. Hübner, J.F., Boissier, O., Bordini, R.H.: From organisation specification to normative programming in multi-agent organisations. In: International Workshop on Computational Logic in Multi-Agent Systems. pp. 117–134. Springer (2010)
19. Hübner, J.F., Boissier, O., Bordini, R.H.: A normative programming language for multi-agent organisations. Annals of Mathematics and Artificial Intelligence **62**(1), 27–53 (2011)
20. Hübner, J.F., Boissier, O., Kitio, R., Ricci, A.: Instrumenting multi-agent organisations with organisational artifacts and agents. Autonomous Agents and Multi-Agent Systems **20**(3), 369–400 (5 2010)
21. 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). https://doi.org/10.1504/IJAOSE.2007.016266, http://dx.doi.org/10.1504/IJAOSE.2007.016266
22. Hübner, J.F., Sichman, J.S., Boissier, O.: A model for the structural, functional, and deontic specification of organizations in multiagent systems. In: Bittencourt, G., Ramalho, G.L. (eds.) Advances in Artificial Intelligence. pp. 118–128. Springer Berlin Heidelberg (2002)
23. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., et al.: Web services business process execution language version 2.0. OASIS standard **11**(120), 5 (2007)
24. Klein, M., Dellarocas, C.: Exception handling in agent systems. In: AGENTS '99 (1999)

25. Klein, M., Dellarocas, C.: A knowledge-based approach to handling exceptions in workflow systems. Computer Supported Cooperative Work (CSCW) **9**(3-4), 399–412 (2000)
26. Mallya, A.U., Singh, M.P.: Modeling exceptions via commitment protocols. In: Proc. of the 4th Int. J. Conf. on Auton. Agents and Multiagent Systems. pp. 122–129. AAMAS '05, ACM (2005)
27. Meyer, B.: Object-oriented software construction, vol. 2. Prentice Hall New York (1988)
28. Platon, E.: Modeling exception management in multi-agent systems. Ph.D. thesis, Université Pierre et Marie Curie, France (2007)
29. Platon, E., Sabouret, N., Honiden, S.: Challenges for exception handling in multi-agent systems. In: Choren, R., Garcia, A., Giese, H., Leung, H.f., Lucena, C., Romanovsky, A. (eds.) Software Engineering for Multi-Agent Systems V. pp. 41–56. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
30. Platon, E., Sabouret, N., Honiden, S.: A definition of exceptions in agent-oriented computing. In: O'Hare, G.M.P., Ricci, A., O'Grady, M.J., Dikenelli, O. (eds.) Engineering Societies in the Agents World VII. pp. 161–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
31. Platon, E., Sabouret, N., Honiden, S.: An architecture for exception management in multiagent systems. International Journal of Agent-Oriented Software Engineering **2**(3), 267–289 (2008)
32. Sommerville, I.: Models for Responsibility Assignment, pp. 165–186. Springer London, London (2007)
33. Sommerville, I., Storer, T., Lock, R.: Responsibility modelling for civil emergency planning. Risk Management **11**(3), 179–207 (2009)