Accountability and Agents for Engineering Business Processes

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. Business processes realize a business goal by coordinating the tasks undertaken by multiple interacting parties. Even if it is possible to monitor the execution of such complex distributed process, current approaches do not allow participants to report to the right account taker the causes of the success or failure of their duties. However, as in exception management in programming languages, having such information could enable the account taker to properly handle errors. We claim that an explicit representation of accountability and responsibility assumptions provides the right abstractions to engineer multi-agent systems, that execute such business processes, both at the level of design and at the level of programming. Basing our programming approach on multi-agent organizations, we present two accountability patterns for developing accountable agents. To illustrate this approach we use the JaCaMo multi-agent programming platform.

1 Introduction

Weske [39] defines a business process as "a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal." In general, a business goal is achieved by breaking it up into subgoals, which are distributed to a number of actors. Each actor carries out part of the process, and depends on the collaboration of others to perform its task. One limit of business processes is that they integrate, at the same abstraction level, both the business logic and the interaction logic (message passing). This, on one hand, makes their reuse problematic–whenever different coordination schemas are to be enacted the business process must be revised. On the other hand, since message exchanges lie at the level of data, it is difficult to assess the correctness of individual processes in isolation.

Multiagent Systems (MAS), and in particular models for MAS organizations, are promising candidates to supply the right abstractions to keep processes linked together in a way that allows reasoning about the correctness of the overall system in terms of goals, rather than of messages. However, agent organizations are still lacking of a systematic way to treat exceptions at execution time. The point is that when an exception does occur, the agent which is apt to handle it (or which is interested to know), may be not the same agent who detects the exception. To make the overall system robust, the exception should be reported to the agent with the proper means for treating it.

In [2] a proposal was made to use accountability and responsibility relationships to state the rights and duties of agents in the organisation, given the specification of a normative structures. From this understanding we define what it means for an agent to be accountable when taking responsibilities in the execution of part of a business process. That is, we address the notion of accountability from a computational perspective and study how it can be obtained as a design property [5]. In the following we show how robustness can be achieved via accountability and responsibility relationships, and we use these concepts as tools to systematize and guide the design and development of the agents. We then exemplify how such concepts can be engineered in the particular context of a JaCaMo multi-agent system where agents execute under a normative organization expressing business process as accountability and responsibility relations among agents.

2 Responsibility and Accountability as Engineering Concepts

Why Should Current BPM and MAO Be Better Engineered

In this paper we consider the Incident Management scenario (from the BPMN examples by the OMG [31]), see Figure 1. The case models the interaction between a customer and a company for the management of a problem reported by the customer. It involves several actors. The customer reports the problem to a Key Account Manager who, on the basis of her experience, can either resolve the problem directly or ask for the intervention of first-level support. The problem is, then, recursively treated at different support levels until, in the worst case, it is reported to the software developer. Generally, the business aim of the process (to solve the reported problem) is decomposed and can be distributed over five BPMN processes, whose execution requires interaction and coordination, realized through message exchange. Noticeably, as always with business processes, the way in which goals are achieved matters, so the agents that will participate into the organization are expected not only to fulfill their assigned goals but also to respect the business process: from an organizational perspective, the "goal" is that the process takes place [1].

Limitations/Problems in Current BPM and MAO

Goal distribution over a group of processes bears strong similarities with proposals from research on MAS organizations. For what concerns software modularity, both business processes and MAS organizations suffer from some limitations and drawbacks; in particular, by focussing merely on the achievement of the assigned sub-goals, agents lose sight of the overall process, and *ignore* the place of their achievement has within the organization. So, for instance, in BPMN the relationships between the actors are just loosely modeled via message exchange, and there is no explicit representation of the responsibilities each of them takes as a party of the interaction nor of the legitimate expectations each actor has of the others. The relationship between each level of support and the following one, in the example, is emblematic: when a request of support is made, an answer containing some kind of feedback is expected in order to proceed.

However, since processes are independent, one cannot give for granted that another will answer. It follows that when a process does not answer, the waiting one may get stuck indefinitely.

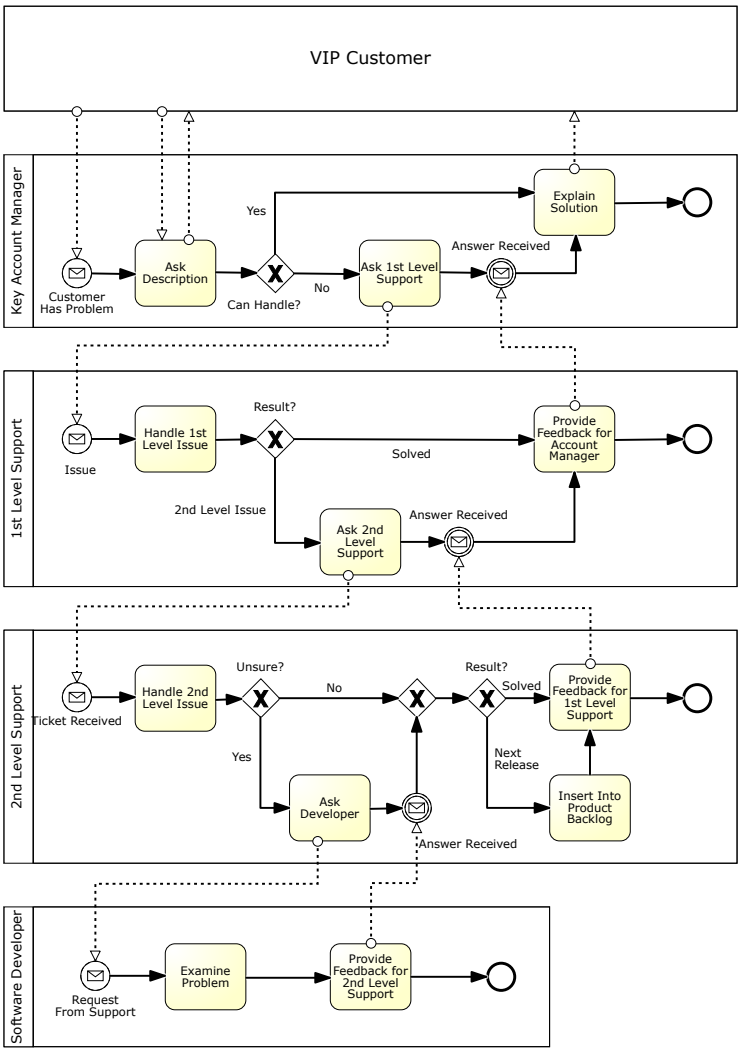

Fig. 1: The incident management BPMN diagram.

Similarly, MAO (see, e.g., [12, 16]) allow structuring complex, organizational goal via a functional decomposition, whereby subgoals are assigned to agents. The coordinated execution of subgoals is often supported by a normative specification, with which the organization issues obligations towards the agents (e.g. [15, 20, 14, 9]). However, agents may have the capability of achieving the assigned goals but in ways that do not fit into the process specification and, more importantly, when agents fail, the organization has no explicit mechanism for sorting out what occurred, for a redress.

Even if agent organizations solve part of the limitation of BPMN, what is actually missing is the agents' awareness of their part in the organization, not only in terms of the goals assigned to them but also (and equally important) in terms of the relationships they have with the others, of their mutual dependences, and, more broadly, of the dependence of the organization on its members for what concerns the *realization of the business process*. We claim that the notions of *responsibility* and *accountability* serve this purpose in an intuitive, yet effective way. A first conceptualization of how these notions can be used in the context of distributed processes is discussed in [7], here we discuss more practical, programming aspects.

Responsibility and Accountability

According to Dubnick [17], accountability "emerges as a primary characteristic of governance where there is a sense of agreement and certainty about the legitimacy of expectations between the community members." So, within an institutional frame, accountability manifests as rules, through which authority is "controlled" so that it is exercised in appropriate ways. In human organizations, it amounts to the enactment of mechanisms for dealing with expectations/uncertainty. In complex task environments where multiple, diverse and conflicting expectations arise, it is a means for managing an otherwise chaotic situation. Further on this line [22], accountability implies that some actors have the right to hold other actors to a set of standards, to judge whether they have fulfilled their responsibilities in light of these standards, and to impose sanctions if they determine that these responsibilities have not been met. They explain that accountability presupposes a relationship between power-wielders and those holding them accountable, where there is a general recognition of the legitimacy of (1) the operative standards for accountability and (2) the authority of the parties to the relationship (one to exercise particular powers and the other to hold them to account).

Concerning responsibility, [38] proposes an ontology relating six different responsibility concepts (capacity, causal, role, outcome, virtue, and liability), that capture: doing the right thing, having duties, an outcome being ascribable to someone, a condition that produced something, the capacity to understand and decide what to do, something being legally attributable. In the context of Information Systems (in particular, access rights models and rights engineering methods), the meta-model ReMMO [19] represents responsibility as a unique charge assigned to an agent, and in the cited literature most of the authors acknowledge that responsibility aims at conferring one or more obligation(s) to an actor (the responsibility owner). As a consequence, this causes a moral or formal duty, in the mind of the responsibility owner, to justify the performance of the obligation to someone else, by virtue of its accountability.

Business processes, represent an agreed behavior, introduce expectations on the behavior of the interacting parties, and require some kind of governance in order for the process to be enacted. Thus, they show all the characteristics of accountability settings, but the lack of an adequate representation obfuscates the accountability [30], which results hidden into some kind of collective responsibility –often taking the shape of the so called "many hands problem". As a consequence, the governance of the system is compromised as well as its functioning as a whole. As Thompson [37] explains, typically adopted solutions for avoiding the many hands problem, like applying hierarchical or

collective forms of responsibility, are wanting, and personal responsibility approaches, based on some weak causal connection between an individual and the event, should be preferred. It is worth noting that accountability and responsibility are not primitive concepts. Rather, they are properties that emerge in carefully designed software systems. This means that when we use accountability/responsibility as engineering tools, we actually constrain the ways in which software is designed and developed.

3 Engineering MAO with Accountability/Responsibility

JaCaMo Organisation Model

JaCaMo [10] is a conceptual model and programming platform that integrates agents, environments and organizations. A MAS in JaCaMo consists of an agent organization, realized through $\mathcal{M}$OISE [26], involving Jason [11] autonomous agents, working in a shared, artifact-based environment, programmed in CArtAgO [34]. A Jason agent consists of a set of plans, each having the structure $triggering\_event : \langle context \rangle \leftarrow \langle body \rangle$. On occurrence of *triggering_event* (belief/goal addition or deletion), under the circumstances given by *context*, the course of action expressed by *body* should be taken.

$\mathcal{M}$OISE includes an organization modeling language and an organization management infrastructure [25]. The specification of an organization is decomposed into three dimensions. The *structural* dimension specifies roles, groups and links between roles in the organization. The *functional* dimension is composed of one (or more) scheme capturing how the global organizational goal is decomposed into subgoals, and how subgoals are grouped in sets, called missions, to be distributed to the agents. The *normative* dimension binds the two previous dimensions by specifying roles' permissions and obligations for missions.

JaCaMo provides various kinds of organizational artifacts that allow encoding the state and behavior of the organization, in terms of groups, schemes and normative states. Obligations are issued on the basis of a normative program, written in NOPL [24]. Norms have the form $id : \phi \rightarrow \psi$, where id is a unique identifier of the norm; ϕ is a formula that determines the activation condition for the norm; and ψ is the consequence of the activation of the norm (either a failure or the generation of an obligation). Obligations, thus, have a well-defined lifecycle. Once created, an obligation is *active*. It becomes *fulfilled* when the agent, to which the obligation is directed, brings about the state of the world specified by the obligation before a given deadline. An obligation is *unfulfilled* when the agent does not bring it about before the deadline. When the condition ϕ does not hold anymore, the state becomes *inactive*.

Accountability/Responsibility Specifications in the JaCaMo Organisation Model

As introduced in [2], we denote by $\mathsf{R}(x, q)$ and $\mathsf{A}(x, y, r, u)$ responsibility and accountability relationships, respectively. $R(x, q)$ expresses an expectation on any agent playing role x on pursuing condition q (x is entitled and should have the capabilities of bringing about q). Instead, $\mathsf{A}(x, y, r, u)$ expresses that x, the account-giver (a-giver), is

accountable towards y, the account-taker (a-taker), for the condition u when the condition r (*context*) holds. We see u in the context of r as the agreed standard which brings about expectations inside the organization.

Since the proposal is set into the JaCaMo framework [10], the coordinated execution of the agents is regulated by obligations, issued by the organization. Agents, however, are autonomous and their fulfillment of the obligations cannot be given for granted. In [2], it is therefore proposed to improve the specification of an organization by complementing the functional decomposition of the organizational goal with a set of accountability and responsibility specifications. Precisely, accountability relationships can be collected in a set $\mathbf{A}$, called an *accountability specification*. The organization designer will generally specify a set of accountability specifications which is denoted by $\mathbb{A}$. In the following, we show in JaCaMo how accountability and responsibility are taken into account at *design time*. In particular, we discuss a programming pattern for accountable agents, that is, agents that provide an account of their conduct both when they succeed in achieving their goals, and when, for some reason, they fail in the attempt.

To specify the execution conditions that are object of accountability and responsibility, we use the event-based linear logic called *precedence logic* [35]. Such a language allows modeling complex expressions, under the responsibility of many agents, whose execution needs to be coordinated. The interpretation deals with occurrences of events along runs (i.e., sequence of instanced events). Event occurrences are assumed non-repeating and persistent: once an event has occurred, it has occurred forever. The logic has three primary operators: '$\vee$' (choice), '$\wedge$' (concurrence), and '$\cdot$' (before). The *before* operator constrains the order with which two events must occur: $a \cdot b$ means that a must occur before b, but not necessarily one immediately after the other. If e be an event, $\overline{e}$ (the complement of e) is also an event. Initially, neither e nor $\overline{e}$ hold. On any run, either of the two may occur, not both. Complementary events allow specifying situations in which an expected event e does not occur, either because of the occurrence of an opposite event, or because of the expiration of a time deadline.

Residuation, inspired by [29, 35], allows to track the progression of temporal logic expressions, hopefully arriving to completion of their execution. The *residual* of a temporal expression q with respect to an event e, denoted as q/e, is the remainder temporal expression that would be left over when e occurs, and whose satisfaction would guarantee the satisfaction of the original temporal expression q. Residual can be calculated by means of a set of rewrite rules. The following equations are due to Singh [35, 29]. Here, r is a sequence expression, and e is an event or $\top$. Below, Γ_u is the set of literals and their complements mentioned in u. Thus, for instance, $\Gamma_e = \{e, \overline{e}\} = \Gamma_{\overline{e}}$ and $\Gamma_{e \cdot f} = \{e, \overline{e}, f, \overline{f}\}$. We have that:

$$
\begin{array}{ll}
0/e \doteq 0 & \top/e \doteq \top \\
(r \wedge u)/e \doteq ((r/e) \wedge (u/e)) & (r \vee u)/e \doteq ((r/e) \vee (u/e)) \\
(e \cdot r)/e \doteq r, \text{ if } e \notin \Gamma_r & (e' \cdot r)/e \doteq 0, \text{ if } e \in \Gamma_r \\
r/e \doteq r, \text{ if } e \notin \Gamma_r & (\overline{e} \cdot r)/e \doteq 0
\end{array}
$$

Using the terminology in [2], we say that an event e is *relevant* to a temporal expression p if that event is involved in p, i.e. $p/e \not\equiv p$. Let us denote by $\boldsymbol{e}$ a sequence $e_1, e_2, \ldots, e_n$ of events. We extend the notion of residual of a temporal expression q to

a sequence of events $\boldsymbol{e}$ as follows: $q/\boldsymbol{e} = (\ldots((q/e_1)/e_2)/\ldots)/e_n$. If $q/\boldsymbol{e} \equiv \top$ and all events in $\boldsymbol{e}$ are relevant to q, we say that the sequence $\boldsymbol{e}$ is an *actualization* of the temporal expression q (denoted by $\widehat{q}$).

Agent Programming Patterns

Given a set of accountability specifications $\mathbb{A}$, and a set of responsibility assumptions $\mathbf{R}$ (responsibility distribution), the organization is properly specified when the *accountability fitting* "$\mathbf{R}$ *fits* $\mathbb{A}$" (denoted by $\mathbf{R} \rightsquigarrow \mathbb{A}$) holds. This happens if $\exists\ \mathbf{A} \in \mathbb{A}$ such that $\forall\ \mathsf{A}(x, y, r, u) \in \mathbf{A}$, $\exists\ \mathsf{R}(x, q) \in \mathbf{R}$ such that, for some actualization $\widehat{q}$, $(u/r)/\widehat{q} \equiv \top$.

Fitting has a relevant impact on organization design: When $\mathbf{R} \rightsquigarrow \mathbb{A}$ holds, any set of agents playing roles into the organization (consistently with $\mathbf{R}$ and one accountability specification $\mathbf{A} \in \mathbb{A}$) can actually accomplish the organizational goal, see [2]. Moreover, fitting provides a guide for developing agents that are *accountable by design*, because it expresses what an agent is engaged to achieve, by fulfilling its responsibilities, and how this achievement is related to that process which is the goal of the organization (through accountability). In other words, we claim that $\mathbf{R} \rightsquigarrow \mathbb{A}$ provides a specification the agents must explicitly conform to, when enacting organizational roles.

When an agent enacts some role in an organization, it declares to be aware of all the responsibilities that come with that role, and by accepting them it declares to adhere to the fitting exposed by the organization itself. That is, the accountability fitting exposed by an organization specifies the requirements that agents, willing to play roles in that organization, must satisfy.

As seen in Section 2, when an agent accepts a responsibility it accepts to *account for* the achievement, or failure, of some state of interest. In our metaphor, thus, an agent acts with the aim of preparing the account it should provide. In this way, we reify the cited "sense of agreement and certainty about the legitimacy of expectations between the community members" which otherwise remains implicit both in business processes and in MAS organizations. Leveraging these concepts for developing agents provides interesting advantages from a software engineering point of view. We now introduce two programming patterns that allow realizing accountable agents, but before we need to identify the portion of fitting involving each single individual.

Definition 1. *Given the fitting $\boldsymbol{R} \rightsquigarrow \mathbb{A}$, and a role x in its scope, the projection of the fitting over role x is defined as $\boldsymbol{R}_x \rightsquigarrow \boldsymbol{A}_x$ where $\boldsymbol{R}_x \equiv \{R(x, q) | R(x, q) \in \boldsymbol{R}\}$, and $\boldsymbol{A}_x \equiv \{A(x, y, r, u) | A(x, y, r, u) \in \boldsymbol{A}\}$, and where for every $A(x, y, r, u) \in \boldsymbol{A}_x$, there is $R(x, q) \in \boldsymbol{R}_x$, such that $(u/r)/\widehat{q} \equiv \top$ holds for some actualization $\widehat{q}$ of q.*

Indeed, since accountabilities and responsibilities imply some obligations [22], we can think of realizing them in JaCaMo by relying on the deontic primitive elements that such framework provides. In other words, the fitting projection over role x can then be mapped into a number of Jason plans of the agent playing role x by way of the following patterns, expressed in AgentSpeak(ER). AgentSpeak(ER) [33] extends Jason by introducing two types of plans: g-plans encapsulate the strategy for achieving a goal and can be further structured into sub-plans. Besides triggering events and contexts, g-plans include a goal condition, specifying until when the agent should keep pursuing

the goal. Instead, e-plans, defined in the scope of a g-plan, embody the reactive behavior to adopt while pursuing the g-plan's goal.

Definition 2 (Pattern Specification). *The fitting relationship represented by each pair* $\langle R(x,q),\ A(x,y,\ r,u)\rangle$ *in* $\boldsymbol{R}_x \rightsquigarrow \boldsymbol{A}_x$*, is mapped into an AgentSpeak(ER) g-plan according to the following pattern:*

+!be_accountable(x, y, q) <: drop_fitting(x, y, q) {
  +obligation(x, q) **:** $r \wedge c$
    <– $body_q$. ***Well-Doing e-plan***
  +oblUnfulfilled(x, q) **:** $r \wedge c'$
    <– $body_f$. ***Wrong-Doing e-plan***
}

Such that: (1) $body_q$ *satisfies the* fitting-adherence *condition (see below); (2)* $body_f$ *includes the sending of an explanation for the failure from* x *to* y*.*

The agent will perceive, through the identity that is provided by the organizational role it plays, certain events as events it should tackle through some behavior of its own, but it will also be aware of its social position both (1) by knowing some other agent will have the right, under certain conditions, to ask for an account and (2) by including specific behavior for building such an account. The two e-plans encode the proactive behavior of an agent assuming a responsibility. From that moment on, and until the responsibility is not dropped, the agent starts reacting to obligations in accordance to the accountability relationship specified in the fitting.

Well-doing e-plan. The first e-plan is triggered when the specified obligation is issued by the normative organization. That will be the usual obligation a Jason agent receives from the $\mathcal{M}$OISE organization when it is time to pursue a particular goal. The context expression, $r \wedge c$, is satisfied when condition r activating the agent accountability holds together with some possibly empty condition c: a local condition that encodes the possibility for the agent to have multiple well-doing e-plans to react to the same obligation, i.e. multiple ways to achieve a same result in different (local) circumstances (e.g., a manager could decide to handle a task directly or delegate it to an employee). Condition c allows the developer to discriminate between these alternatives, if any. It's worth noting that if multiple alternative e-plans with different c are present, the developer must take care of defining such conditions so that for each obligation issued, at least one e-plan is always triggered. Due to the accountability fitting the agent has accepted, the body of the plan(s) ($body_q$) must,then, be such to satisfy the responsibility assumption represented by the pair $\langle R(x,q), A(x,y,r,u)\rangle$. That is, the plan body has to satisfy the following *fitting-adherence* condition.

Definition 3. *(Fitting-adherence) Let* $[body_q]_u$ *denote the set of sequences of events generated by the execution of* $body_q$*, restricted to the events that are relevant for the progression of* u*.* $body_q$ *satisfies the* fitting-adherence *condition if:* $\exists$ *sequence* $s \in [body_q]_u$ *such that* $s \equiv \widehat{q}$ *and* $(u/r)/\widehat{q} \equiv \top$.

Note that fitting adherence requires the agent to be just able to activate at least one actualization s of q, not all of them. In other words, the agent needs to be able to

perform at least one of the possibly many ways for carrying out q. The rationale is that any actualization of q generates a sequence of events that brings the condition u/r to $\top$; hence, it is sufficient for an agent to implement one actualization in order to meet its responsibility. As we have discussed above, an accountable agent provides an account of its conduct. In our framework, the account of a well-doing pattern is immediately given by the agent's behavior itself. Following Garfinkel [21], we consider that the account, or proof, that the agent behaved correctly is evident by how the agent has operated in the environment. This means that the obligation to give an account for the satisfaction of an obligation is implicitly resolved by satisfying the very same obligation. Thus, there is not the need to capture the obligation to give an account explicitly. When, however, it is not possible to see the agent's operations as a proof, an explicit account should be provided also for the well-doing case. This, for instance, happens when there is the need of reporting facts that occur in one context, but are meaningful also in others where they are not directly observable by the involved parties, see e.g., [4]. It is also the case in which an agent's behavior requires some certification for having been performed up to some standard. It is interesting to note that the accountability fitting is not only a functional specification of the organization, but it also specifies the "good" behavior of the agents. It is in fact this characteristic that justifies our programming patterns, and that it is not captured in standard JaCaMo.

Wrong-doing e-plan. The second pattern allows the agent to provide an account also when the agent does not complete a task, for some reason. The triggering event, *oblUnfulfilled*, is generated by the $\mathcal{M}$OISE organization when a previously issued obligation has been left unsatisfied. The context of the pattern is again a condition that is true when the accountability is activated (i.e., r holds), and when some local condition c' is satisfied. $body_f$, this time, has to produce an account about the failure. We can think of such an account as an explanation that the agent produces so that another agent, possibly the a-taker y, can use it to resume the execution, thus managing the exception. The correct use of the pattern guarantees, by design, that exceptional events, when occurring, are reported to the agents who can handle them properly. Accountability fulfills this purpose because, by nature, it brings about an obligation on the a-giver to give an account of what it does. The account, then, can be used by the a-taker to recover, when possible, from the exceptional situation. Under this respect, the account should be provided in terms that can be understood by all the interested agents in the organization. This aspect, however, is strongly domain dependent. As well as in the positive pattern, the agent will produce an account by modifying its environment in a way that is meaningful for the agents that have to capture and interpret it.Along this line, a promising approach to the synthesis of an account is discussed in [13].

4 JaCaMo Accountable Agents

In JaCaMo, the state of an organization is encoded in terms of group instances (i.e., which agents are playing which roles) and scheme instances (i.e., which goals were achieved, which ones are ready to be pursued, etc). Notably, such scheme instances are declarative in nature: they only specify which (sub)goals should be achieved, and in

which order, but they do not specify how. By exploiting these instances, the organization issues proper obligations to role players (i.e., agents). The agents are autonomous in the way they satisfy their obligations, and may also use artifacts that are not globally accessible to all the involved agents. They are, however, held to notify the organization about the completion of their task (see the use of *goalAchieved* messages in Section 4). By capturing these messages, the organization traces the subgoals that have been completed and determine the next obligations to be issued.

The engineering of $\mathcal{M}$OISE accountable organizations involves the following steps[3]:

1. Each process is mapped to a role in the organization;
2. A scheme representing the overall process goal is defined; the successful execution of such a scheme corresponds to the achievement of the process goal;
3. For each activity to be performed in sequence, a corresponding subgoal is added to the scheme, by means of the corresponding operator;
4. For each structured block including a concurrent execution, the corresponding goals, grouped together via the parallel operator, are added to the scheme;
5. If a choice is present inside a process, a number of schemes should be defined, representing the possible courses of action. These schemes are to be instantiated dynamically by the agents, depending on their internal choices. Their execution concurs to the progression of the already present ones.

By applying the steps above to Incident Management we obtain the following. For what concerns the structural specification, five roles are identified, all belonging to a single group: customer (c), key account manager (*am*), first level support (*fls*), second level support (*sls*), and developer (*dev*). For each problem to manage, we assume there will be exactly one agent playing each role. Moreover, we use the label *company* to identify the owner of the specified organization.

The overall organizational goal is distributed into a set of schemes, part of which is reported in Figure 2. The scheme on top involves c and *am*, and is instantiated by the customer c when some need arises. In other terms, the scheme instantiation corresponds to the occurrence of a *report-problem*$_c$ event. When a problem is reported, the account manager is expected to perform *ask-description*$_{am}$ (ask for a description of the problem), and c to send what requested (*send-description*$_c$). Then, *am* should provide a solution: to this aim, it can take two alternatives courses of action, both leading to the same join point of the BPMN diagram. Each path amounts to a new scheme, that will be instantiated by the agent depending on the choice made. Only by completing the execution of the selected scheme the outer scheme will progress. Here, the first alternative amounts to the case in which *am* can handle the problem directly, and does not require the execution of any action before the join point. For this reason, the corresponding scheme would be empty and we omit it. The second scheme, instead, is instantiated when am cannot handle the problem directly. Thus, it will, first, make a request (*ask-support-fls*$_{am}$) that *fls* will either manage directly or will involve the next level of support by instantiating a further scheme. It is worth noting that scheme instantiation is not used just to tackle situations in which one agent needs to interact with

[3] Here we restrict our attention to the translation of structured blocks (see [18]) into social schemes. The presence of a single exit point is necessary to ensure the proper completion of the $\mathcal{M}$OISE schemes.

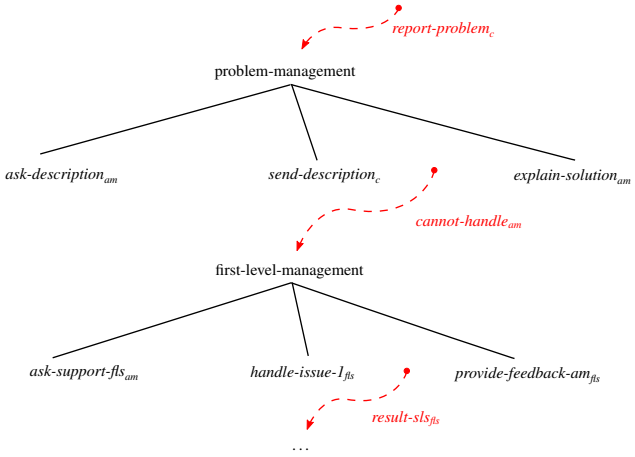

Fig. 2: Part of the functional decomposition of Incident Management.

$\boldsymbol{a_1}$: A(am, c, *report-problem*$_c$, *report-problem*$_c$ · *ask-description*$_{am}$)
$\boldsymbol{a_2}$: A(am, c, *report-problem*$_c$ · *ask-description*$_{am}$ · *send-description*$_c$, *report-problem*$_c$ · *ask-description*$_{am}$ · ·*send-description*$_c$ · *explain-solution*$_{am}$)
$\boldsymbol{a_3}$: A(am, *company*, *report-problem*$_c$, *report-problem*$_c$ · *ask-description*$_{am}$)
$\boldsymbol{a_4}$: A(am, *company*, *report-problem*$_c$ · *ask-description*$_{am}$ · *send-description*$_c$, *report-problem*$_c$ · *ask-description*$_{am}$ · ·*send-description*$_c$ · *explain-solution*$_{am}$)
$\boldsymbol{a_5}$: A(am, *company*, *report-problem*$_c$ · *ask-description*$_{am}$ · *send-description*$_c$ · *cannot-handle*$_{am}$, *report-problem*$_c$ · ·*ask-description*$_{am}$ · *send-description*$_c$ · *cannot-handle*$_{am}$ · *ask-support-fls*$_{am}$)
...

$\boldsymbol{r_1}$: R(am, *ask-description*$_{am}$) $\boldsymbol{r_2}$: R(am, *explain-solution*$_{am}$) $\boldsymbol{r_3}$: R(am, *ask-support-fls*$_{am}$)
...

Fig. 3: Excerpt of the accountability specification and responsibility distribution for the *Incident Management* scenario.

others for some aim, but it may also occur when a process has internal choices. This depends on whether the single branches are subject to accountabilities or not, that is, whether the agent should not only achieve the goal, but also stick to the specified process in doing so. This is, for instance, the case of the second choice in the Second Level Support process (see "Result?" in Figure 1).

Interestingly, the instantiation "on-the-fly" of a scheme can be seen as a form of *planning autonomy*: "This type of autonomy dictates if an agent is able (or unable) to create, choose or modify plans to achieve a specific goal" [27]. The integration of this type of autonomy into an organizational model (i.e., $\mathcal{M}$OISE) discussed in [27] opens interesting perspectives in the modeling of BPMN processes for our accountable agents.

Figure 3 reports an excerpt of an accountability specification $\mathbf{A}_{incident}$ for the incident management scenario. Accountabilities a_1-a_5, in particular concern am as a-taker. The first accountability a_1 states that *am* is accountable towards c for asking for a de-

scription, only after a problem is reported. To guarantee this strict ordering, the event *report-problem*$_c$ appears both as the antecedent condition and as a prefix of the consequent condition, preceding *ask-description*$_{am}$ (the same structure is used throughout the subsequent relationships). This requirement means that c can legitimately expect that, by reporting a problem to *am*, it will be asked for a description of the problem. The second accountability, similar to the previous one, states that once the description of the problem has been provided, *am* must, in the end, explain the solution, no matter what happens in the meanwhile.

Accountabilities a_3-a_5 encode the fact that all workers, inside the incident management company, are expected to stick to the process specification. For this reason, these accountabilities do not include the customer c, who is not an employee of the company. Indeed, in the BPMN representation the customer process is collapsed. The only requirement that should be captured by the accountability specification concerns the proper interaction with *am*. In a_4, *explain-solution*$_{am}$ is the exit point of the structured block beginning with the XOR gateway (see Figure 1). The accountability means that, no matter what path is chosen, the agent must account (either positively or negatively) about the achievement of that task. This accountability is, then, complemented with a_5, which states that if *am* decides it cannot handle the problem directly (*cannot-handle*$_{am}$), then it will execute *ask-support-fls*$_{am}$. In a way, *cannot-handle*$_{am}$ manifests the internal choice made by the agent, and will lead to the instantiation of the second social scheme discussed above. These five accountability, a_1-a_5, completely characterize the *am* agent.

With the accountability specification $\mathbf{A}_{incident}$ as a basis, the designer can identify a suitable responsibility distribution which fits it. An excerpt of an acceptable one, w.r.t. *am*, is reported in Figure 3. It is easy to verify that for each $a_i \in \mathbf{A}_{am}$ there is a $r_j \in \mathbf{R}_{am}$ which fits it. For instance, if we consider a_1 and r_1, we have that: $(\textit{report-problem}_c \cdot \textit{ask-description}_{am})/\textit{report-problem}_c/\ \textit{ask-description}_{am} \equiv \top$.

Engineering Accountability Behaviors in Agents

As an illustration, we briefly explain the realization of the key account manager *am* agent. We restrict our attention to $\mathbf{A}_{am} = \{a_1, a_2, a_3, a_4, a_5\}$ and $\mathbf{R}_{am} = \{r_1, r_2, r_3\}$. For each pair in $\mathbf{R}_{am} \rightsquigarrow \mathbf{A}_{am}$, a g-plan must be defined, containing the proper *well-doing* and *wrong-doing* e-plans. Let us consider, in particular, the fitting involving $r_2 \rightsquigarrow a_2$ implemented by the following plans.

```
+!be_accountable(Ag,ATaker,What)

   : .my_name(Ag) &
     (satisfied(sch1,explain_solution) = What |
      done(sch1,explain_solution,Ag) = What) &
     play(ATaker,customer,incident_group)

  <: drop_fitting(Ag,ATaker,What) {

+obligation(Ag,_,What,_)[artifact_id(ArtId)]
   : .my_name(Ag) & (satisfied(sch1,explain_solution) = What |
     done(sch1,explain_solution,Ag)=What) &
     goalState(sch1,ask_description,_,_,satisfied) &
     goalState(sch1,send_description,_,_,satisfied) &
     play(Customer,customer,incident_group) & can_handle(What)
  <- println("Explaining solution...");
     .send(Customer,tell,explain_solution);
```

```
18      goalAchieved(explain_solution)[artifact_id(ArtId)].
19
20  +obligation(Ag,_,What,_)[artifact_id(ArtId)]
21     : .my_name(Ag) & (satisfied(sch1,explain_solution) = What |
22       done(sch1,explain_solution,Ag)=What) &
23       goalState(sch1,ask_description,_,_,satisfied) &
24       goalState(sch1,send_description,_,_,satisfied) &
25       play(Customer,customer,incident_group) &
26       not can_handle(What) &
27       orgArt(OrgArtId) & grArt(GrArtId)
28    <- createScheme(sch2, scheme2, SchArtId)[artifact_id(OrgArtId)];
29       debug(inspector_gui(on))[artifact_id(SchArtId)];
30       focus(SchArtId); addScheme(sch2)[artifact_id(GrArtId)];
31       ?goalState(sch2,provide_feedback_am,_,_,satisfied)[artifact_id(SchArtId)];
32       .send(Customer,tell,explain_solution);
33       goalAchieved(explain_solution)[artifact_id(ArtId)].
34
35  +oblUnfulfilled(O)
36     : .my_name(Ag) &  obligation(Ag,_,What,_) = O &
37       (satisfied(sch1,explain_solution) = What |
38        done(sch1,explain_solution,Ag)=What) &
39       goalState(sch1,ask_description,_,_,satisfied) &
40       goalState(sch1,send_description,_,_,satisfied) &
41       can_handle(What)
42    <- .send(ATaker, tell, operation_failed_error).
43
44  +oblUnfulfilled(O)
45     : .my_name(Ag) &  obligation(Ag,_,What,_) = O &
46       (satisfied(sch1,explain_solution) = What |
47       done(sch1,explain_solution,Ag)=What) &
48       goalState(sch1,ask_description,_,_,satisfied) &
49       goalState(sch1,send_description,_,_,satisfied) &
50       not can_handle(What) &
51       not goalState(sch2,provide_feedback_am,_,_,satisfied) &
52    <- .send(ATaker, tell, please_call_again).
53
54  +cancel-fls-request
55     : oblUnfulfilled(O) &  obligation(_,_,What,_) = O &
56       (satisfied(sch2,provide_feedback_am) = What |
57       done(sch2,provide_feedback_am,_)=What) &
58    <- .send(ATaker, tell, please_call_Again);
59       .drop_all_intentions.
60  ...
61  }
```

The outer g-plan is triggered when the agent proactively decides to adhere to the fitting $r_2 \rightsquigarrow a_2$, thereby becoming accountable for the task. Once triggered, the g-plan will remain active until the agent does not drop the fitting (see Line 8). The plans in braces encode the reactive behavior corresponding to the *well-doing* and *wrong-doing* e-plans specified by the pattern. The first two plans, in particular, realize the *well-doing* part of the pattern. Recalling Definition 2, the plans are triggered as soon as an obligation to explain the solution to the customer's problem is issued (Line 10). The obligation's object (*What*) is the satisfaction of the organizational goal *explain_solution* (Lines 11 and 12) –corresponding to the *explain-solution*$_{am}$ event in Figure 3. Indeed, in JaCaMo, the achievement of an organizational goal fulfills the corresponding obligation. Then, as requested by the pattern, the contexts of both plans must include the conditions specified in a_2. In JaCaMo we represent these conditions in terms of schemes that were instantiated and in terms of organizational goals that were achieved. Considering the fitting-adherence condition, we have that r is *report-problem*$_c$·*ask-description*$_{am}$·*send-description*$_c$, and u is *report-problem*$_c$ · *ask-description*$_{am}$ · *send-description*$_c$ ·

explain-solution$_{am}$. Thus $q \equiv u/r$ is just *explain-solution*$_{am}$. Both plans for *well-doing*, thus, need to include some actions that amount to such an event. In this setting we consider the event *explain-solution*$_{am}$ to be occurred as soon as the corresponding organizational goal is set as achieved. This is trivially true in the example (see Lines 18 and 33). Here, the achievement of the goal is notified to the organization after the construction of the answer to the customer (represented in the code by a simple print in one case, and by the interaction with *fls* in the other). Agent acts by modifying the organizational environment. This leads to the construction of a sequence of facts that constitutes the account for the specific goal.

The presence of two plans triggered by the same obligation reflects the internal choice inside the business process, driven by some local condition. The first plan is executed when *am* can handle the problem directly (Line 15): the solution to the problem is simply sent to the customer and the corresponding organizational goal is marked as achieved. The second plan, instead, is executed when the agent decides to ask for support (Line 26). In this case, before providing a feedback to the customer the agent will create an instance of the second social scheme (Line 28), thereby making its choice (*cannot-handle*$_{am}$) public. The successful scheme completion will provide it with a feedback from *fls* (Line 31): *am* can legitimately expect such a feedback by virtue of accountability a_7, in which it is a-taker. The feedback, in turn, will enable the agent to execute *explain-solution*$_{am}$, by setting the organizational goal as achieved and producing an actualization of the formula in the obligation, as discussed above.

The third and fourth plans, at Lines 35 and 44, instead, deal with the *wrong-doing* part of the pattern. Should, for any reason, the obligation be unfulfilled, the agent, by virtue of its accountabilities, must provide a motivation about the unsatisfaction of the obligation to the account-taker. The plan at Line 35, in particular, is triggered when the obligation is unfulfilled because of a reason that is internal to the *am* agent itself (e.g., the plan at Line 15 was triggered, but not successfully completed). The fourth plan, instead, at Line 44, is triggered when the obligation becomes unfulfilled because *am* is still waiting for a feedback from *sls*. In both cases, a proper message encoding the explanation for the failure is sent to the a-taker (see Lines 42 and 52).

The last plan, at Line 54, in turn exemplifies how *am* behaves as an a-taker when receives the account of a failure from another (a-giver) agent. Specifically, the plan handles a possible failure coming from *fls* raised when it has not satisfied its obligation to provide a feedback. Event `cancel-fls-request` corresponds to the message *fls* sends as an account of such a failure, and *am* handles such a failure by asking the customer to call another time and dropping its current intention(s).

Notably, considering the accountability specification as a requirement, the actual implementation of the system results more robust. The accountable *am* for instance, can be rewritten in BPMN as shown in Figure 4: to satisfy the requirement of being accountable, *am* must be capable, on the one side, of capturing exceptions from other agents (specifically, *fls*), and on the other side, of providing an account to its a-taker (i.e., the customer). The other processes are modified in a similar way.

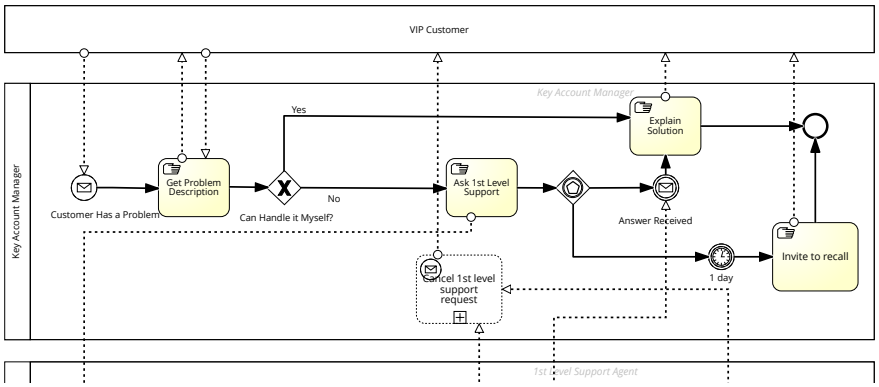

Fig. 4: The accountable process of Account Manager.

5 Conclusions

In this paper we have focused on the development of accountable agents. So, we have discussed how the accountability/responsibility specification of an organization can be mapped into two patterns for programming Jason plans. The two patterns, when applied systematically, bring along positive consequences. First of all, an accountability/responsibility specification provides a programmer with all the relevant information for developing an agent that is aware of the process as characterization of the goal. In fact, while a responsibility distribution is a coverage of the functional decomposition, an accountability specification conveys the programmer how the agent being developed contributes to the process. Hence, the accountabilities provide the programmer with a behavioral specification the agent has to satisfy.

Our approach is specular to [40], where the objective is to determine whether a group of agents can be attributed the responsibility for a given goal. Once the responsibility can be attributed to the agents, their accountability is implicitly modeled in the plan that has be inferred. Here, instead, we aim at developing agents that, by construction, satisfy the organization specification. Indeed, an interesting evolution of the present work goes in the direction of an agent-oriented type checking (see e.g., [3]). Having an explicit model of the organization in terms of accountabilities and responsibilities, it would be possible to mechanize a type checking system that verifies whether, at role enacting time, an agent possesses all the necessary plans for role playing.

The proposal moves MAOs closer to other paradigms where exceptions are handled. In the actor model (e.g., [23]), for instance, when an actor cannot handle an exception, it usually reports the exception to its parent actor, which in turns decides to either handle the exception or report it further. In an agent-based system such a scheme is not directly applicable since agents are independent entities, and rarely are related to each other by a parent-child relationship. In the MAS community, approaches for modeling exceptions in a multi-agent setting have been proposed (see, e.g., [28, 36, 32]). However, no consensus has been reached w.r.t. the usage of such a concept in agent systems. The main problems rise when trying to accomodate the usual exception handling semantics with

the properties of MAS; namely autonomy, openness, heterogeneity, and encapsulation. Accountabilities can fill in this gap: when an obligation is not satisfied, it is reasonable to report the exception to the a-taker (see above example). This is achieved quite naturally with the *Wrong-Doing Pattern*, that allows an agent to provide an account for an unsatisfied obligation.

Commitment-based protocols (e.g., [41]), as well as standard NorMAS [8], provide alternatives for modeling coordination. Roughly speaking, a commitment is a promise that a debtor does in favor to a creditor that in case some antecedent condition is satisfied, the debtor will bring about a consequent condition. When the antecedent holds, the commitment is detached, and amounts to an obligation on the debtor to bring about the consequent. When the consequent is no longer achievable, the commitment is violated. In such a case, the creditor has the right to complain against the debtor, the creditor cannot hold the debtor to provide an explanation. This lack of information hampers both the understanding of what has occurred, and any attempt of recovery from the failure. However, commitments have the power of enforcing accountability when properly used. For instance, the ADOPT protocol [6] establishes an accountability relationship, expressed via a commitment-based protocol, between an organization and its agents.

References

1. Adamo, G., Borgo, S., Di Francescomarino, C., Ghidini, C., Guarino, N.: On the notion of goal in business process models. In: Ghidini, C., Magnini, B., Passerini, A., Traverso, P. (eds.) AI*IA 2018 - Advances in Artificial Intelligence - XVIIth International Conference of the Italian Association for Artificial Intelligence, Trento, Italy, November 20-23, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11298, pp. 139–151. Springer (2018)
2. Baldoni, M., Baroglio, C., Boissier, O., May, K.M., Micalizio, R., Tedeschi, S.: Accountability and responsibility in agent organizations. In: PRIMA 2018: Principles and Practice of Multi-Agent Systems, 21st International Conference. Lecture Notes in Computer Science, vol. 11224, pp. 261–278. Springer (2018)
3. Baldoni, M., Baroglio, C., Capuzzimati, F., Micalizio, R.: Type Checking for Protocol Role Enactments via Commitments. Journal of Autonomous Agents and Multi-Agent Systems **32**(3), 349–386 (May 2018)
4. Baldoni, M., Baroglio, C., Chopra, A.K., Singh, M.P.: Composing and Verifying Commitment-Based Multiagent Protocols. In: Wooldridge, M., Yang, Q. (eds.) Proc. of 24th International Joint Conference on Artificial Intelligence, IJCAI 2015. Buenos Aires, Argentina (July 25th-31th 2015), http://ijcai-15.org/
5. Baldoni, M., Baroglio, C., May, K.M., Micalizio, R., Tedeschi, S.: Computational Accountability. In: Chesani, F., Mello, P., Milano, M. (eds.) Deep Understanding and Reasoning: A challenge for Next-generation Intelligent Agents, URANIA 2016. vol. 1802, pp. 56–62. CEUR, Workshop Proceedings, Genoa, Italy (December 2016), http://ceur-ws.org/Vol-1802/paper8.pdf
6. Baldoni, M., Baroglio, C., May, K.M., Micalizio, R., Tedeschi, S.: Computational Accountability in MAS Organizations with ADOPT. Applied Sciences **8**(4) (2018)
7. Baldoni, M., Baroglio, C., Micalizio, R.: Goal Distribution in Business Process Models. In: Ghedini, C., Magnini, B., Passerini, A., Traverso, P. (eds.) Proc. of 17th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2018). Lecture Notes in Computer Science, vol. 11298, pp. 252–265. Springer, Trento, Italy (2018)

8. Boella, G., van der Torre, L.W.N., Verhagen, H.: Introduction to normative multiagent systems. In: Normative Multi-agent Systems. Dagstuhl Seminar Proceedings, vol. 07122 (2007)
9. 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
10. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming **78**(6), 747–761 (2013)
11. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons (2007)
12. Corkill, D.D., Lesser, V.R.: The use of meta-level control for coordination in distributed p roblem solving network. In: Bundy, A. (ed.) Proceedings of the 8th International Joint Conference on Ar tificial Intelligence (IJCAI'83). pp. 748–756. William Kaufmann, Los Altos, CA (1983)
13. Cranefield, S., Oren, N., Vasconcelos, W.: Accountability for practical reasoning agents. In: AT 2018: 6th International Conference on Agreement Technologies. LNCS, Springer (2018), accepted/In press
14. Dastani, M., Tinnemeier, N.A., Meyer, J.J.C.: A programming language for normative multiagent systems. In: Handbook of Research on Multi-Agent Systems: semantics and dynamics of organizational models, pp. 397–417. IGI Global (2009)
15. Dignum, V.: A model for organizational interaction: based on agents, founded in logic. Ph.D. thesis, Utrecht University (2004), published by SIKS
16. Dignum, V.: Handbook of research on multi-agent systems: Semantics and dynamics of organizational models (2009)
17. Dubnick, M.J., Justice, J.B.: Accounting for accountability (September 2004), https://pdfs.semanticscholar.org/b204/36ed2c186568612f99cb8383711c554e7c70.pdf, annual Meeting of the American Political Science Association
18. Dumas, M., García-Bañuelos, L., Polyvyanyy, A.: Unraveling unstructured process models. In: Mendling, J., Weidlich, M., Weske, M. (eds.) Business Process Modeling Notation. pp. 1–7. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
19. 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)
20. Fornara, N., Viganò, F., Verdicchio, M., Colombetti, M.: Artificial institutions: a model of institutional reality for open multiagent systems. Artificial Intelligence and Law **16**(1), 89–105 (2008). https://doi.org/10.1007/s10506-007-9055-z
21. Garfinkel, H.: Studies in ethnomethodology. Prentice-Hall Inc., Englewood Cliffs, New Jersey (1967)
22. Grant, R.W., Keohane, R.O.: Accountability and Abuses of Power in World Politics. The American Political Science Review **99**(1) (2005)
23. Haller, P., Sommers, F.: Actors in Scala - concurrent programming for the multi-core era. Artima (2011)
24. Hübner, J.F., Boissier, O., Bordini, R.H.: A normative organisation programming language for organisation management infrastructures. In: Padget, J., Artikis, A., Vasconcelos, W., Stathis, K., da Silva, V.T., Matson, E., Polleres, A. (eds.) Coordination, Organizations, Institutions and Norms in Agent Systems V. pp. 114–129. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
25. 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)

26. 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)
27. Maia, A., Sichman, J.S.: Explicit representation of planning autonomy in MOISE organizational model. In: 7th Brazilian Conference on Intelligent Systems, BRACIS 2018, São Paulo, Brazil, October 22-25, 2018. pp. 384–389 (2018)
28. Mallya, A.U., Singh, M.P.: Modeling exceptions via commitment protocols. In: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems. pp. 122–129. AAMAS '05, ACM (2005)
29. Marengo, E., Baldoni, M., Baroglio, C., Chopra, A., Patti, V., Singh, M.: Commitments with regulations: reasoning about safety and control in REGULA. In: Proc. of the 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS). vol. 2, pp. 467–474 (2011)
30. Nissenbaum, H.: Accountability in a computerized society. Science and Engineering Ethics **2**(1), 25–42 (1996)
31. Object Management Group: Bpmn specification - business process model and notation (2018), http://www.bpmn.org/, online, accessed 08/11/2018
32. Platon, E., Sabouret, N., Honiden, S.: An architecture for exception management in multiagent systems. Int. J. Agent-Oriented Softw. Eng. **2**(3), 267–289 (2008)
33. Ricci, A., Bordini, R.H., Hübner, J.F., Collier, R.: AgentSpeak(ER): An Extension of AgentSpeak(L) improving Encapsulation and Reasoning about Goals. In: AAMAS. pp. 2054–2056. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM (2018)
34. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment Programming in CArtAgO, pp. 259–288. Springer US, Boston, MA (2009)
35. Singh, M.P.: Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition. In: The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings. pp. 907–914. ACM (2003)
36. Souchon, F., Dony, C., Urtado, C., Vauttier, S.: Improving exception handling in multi-agent systems. In: Software Engineering for Multi-Agent Systems II. pp. 167–188. Springer Berlin Heidelberg (2004)
37. Thomson, J.J.: Remarks on causation and liability. Philosophy and Public Affairs **13**(2), 101–133 (1984)
38. Vincent, N.A.: Moral Responsibility, Library of Ethics and Applied Philosophy, vol. 27, chap. A Structured Taxonomy of Responsibility Concepts. Springer (2011)
39. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer (2007)
40. Yazdanpanah, V., Dastani, M.: Distant group responsibility in multi-agent systems. In: PRIMA 2016: Princiles and Practice of Multi-Agent Systems - 19th International Conference, Phuket, Thailand, August 22-26, 2016, Proceedings. pp. 261–278 (2016). https://doi.org/10.1007/978-3-319-44832-9_16
41. Yolum, P., Singh, M.P.: Commitment Machines. In: Intelligent Agents VIII, 8th Int. WS, ATAL 2001. LNCS, vol. 2333, pp. 235–247. Springer (2002)