Accountability and Responsibility in Multiagent Organizations for Engineering Business Processes

Matteo Baldoni[1 [0000−0002−9294−0408]], Cristina Baroglio[1 [0000−0002−2070−0616]], Olivier Boissier[2 [0000−0002−2956−0533]], Roberto Micalizio[1 [0000−0001−9336−0651]], and Stefano Tedeschi[1 [0000−0002−9861−390X]]

[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. Given such a distributed nature, Multiagent Organizations (MAO) are a promising paradigm for conceptualizing and implementing business processes. Yet, MAO still lack of a systematic method for reporting to the right agents feedback about success or failure of a task. We claim that an explicit representation of accountability and responsibility assumptions provides the right abstractions to engineer MAO for supporting the execution of business processes. Basing our programming approach on MAO, we present two accountability patterns for developing accountable agents. To illustrate this approach we use the JaCaMo multi-agent programming platform.

1 Introduction

Weske [38] 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 makes their reuse problematic; whenever different coordination schemas are to be enacted the business process must be revised. Moreover, 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 multi-agent organizations (MAO), 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. In order to provide the right support to BPs, however, MAO need to be enriched with a systematic way to properly handle feedback about the execution, that can be provided by the agents as explanation of goal achievement or non-achievement. Such feedback will generally be of interested to (and should be handled by) some agent which is not the one that can produce it. Consequently, that is, for connecting agents in the right way, an appropriate "infrastructure"

needs to be devised. A significant special case of feedback provision and management is exception handling. In this case, the availability of means for reporting the produced feedback (the exception) to an agent that is capable of tackling it, would increase system robustness. Approaches for modeling exceptions in a multiagent setting have been proposed in the literature (see, e.g., [27, 35, 31]) but no consensus has been reached yet on how accommodating the usual exception handling semantics with the peculiar properties of agents, such as autonomy, openness, heterogeneity, and encapsulation.

In this paper we argue that the notions of *accountability* and *responsibility* are useful both to the general purpose of enriching MAOs with a feedback infrastructure, and to the specific purpose of accommodating exception handling.

In [2] a proposal was made to use accountability and responsibility relationships to state the rights and duties of agents in the organization, given the specification of a normative structures. From this understanding, we define what it means for an agent to be accountable when taking on responsibilities in the execution of part of a business process. That is, we address the notion of accountability from a computational perspective and study its role as a design property [5].

In the following 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 introduced in multi-agent systems realized in JaCaMo, where agents will execute under a normative organization expressing a business process as accountability and responsibility relationships among agents. We use, as a reference example, a revisited version of the Incident Management case from the BPMN examples by the OMG [30]. The implementation is available at `https://di.unito.it/incident`.

2 Enhancing MAOs to Better Support BPs

The Incident Management case [30] (Figure 1), that we use as a running example, models the interaction between a customer and a company for the management of a problem that was reported by the customer. It involves several actors. The Customer reports the problem to a Key Account Manager who, based on her experience, can either solve the problem directly or ask for the intervention of First-level Support. The problem can, then, be recursively treated by different support levels until, in the worst case, it is reported to the software developer.

Goal distribution over a group of processes bears strong similarities with proposals from research on MAO. In the Incident Management example, the business aim of the process (to solve the reported problem) is decomposed and distributed over up to five BPMN processes, whose execution requires interaction and coordination–realized in this case through message exchange. 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. Indeed, from an organizational perspective, the "goal" is that the process takes place [1]. As Figure 1 shows, the case includes treatment of anomalous situations, in terms of message passing. For instance, an issue at the level of Software Developer Support is propagated upwards towards the Customer causing certain activities to occur.

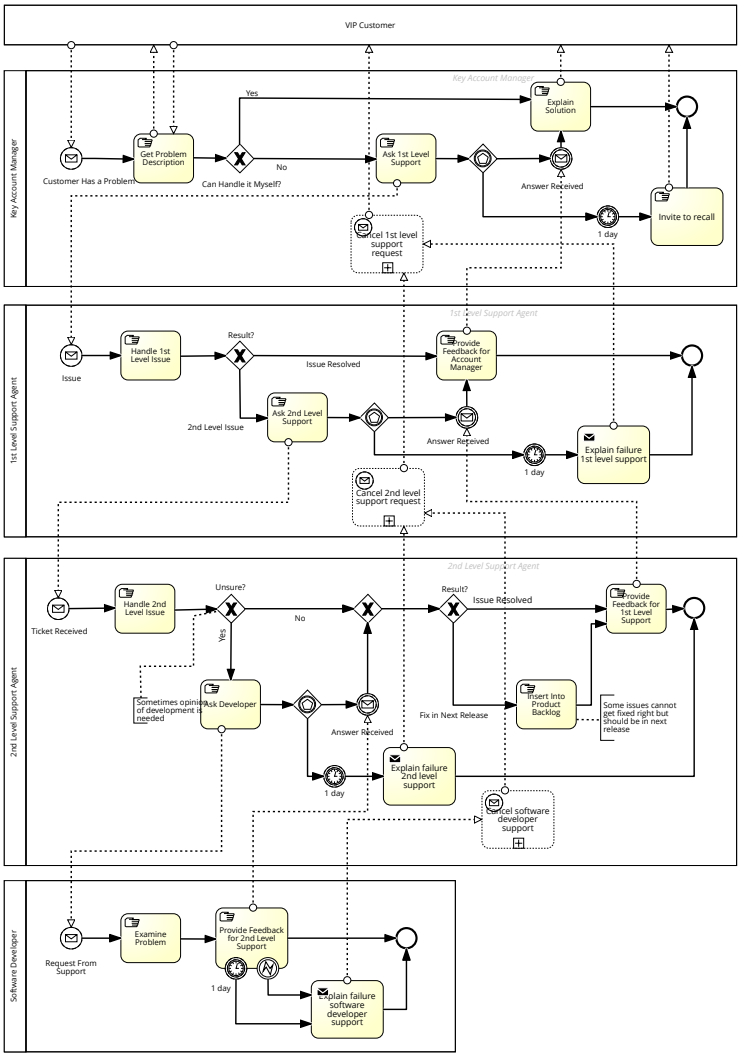

Fig. 1: The incident management BPMN diagram enriched with exception management.

One common limitation of the kind of modularity implemented both by BPs and by MAOs is that the overall process structure of the goal is intended mainly as a way for constraining the agents autonomy, and not as information provided to support the agents in their work. In particular, MAOs (see, e.g., [12, 16]) allow structuring complex organizational goals by functional decomposition, assigning subgoals to the 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, 19, 14, 9]). However, by focusing merely on the achievement of the assigned sub-goals, agents loose sight of the overall process, and ignore the place their achievement has within the or-

ganization. Moreover, agents may have the capability of achieving the assigned goals but in ways that do not fit into the process specification and, importantly, in presence of anomalous situations, the organization has no explicit mechanism for sorting out what occurred, for a redress. On the other hand, in BPMN the relationships between the actors are just *loosely* modeled via message exchange, there is no explicit representation of the legitimate expectations each actor has about the others, and there is no notion of responsibility.

So, even if MAOs solve part of the limits 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 dependencies, 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 [21], 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, [37] 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 [18] 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 show all the characteristics of accountability settings: they represent an agreed behavior, they involve tasks the interacting parties should take care of, they introduce expectations on how they will act, and require some kind of governance in order for the process to be enacted. However, the lack of an adequate representation of the involved relationships obfuscates the accountability [29] (that results hidden into some kind of collective responsibility), possibly compromising the functioning of the system as a whole or its governance. As Thompson [36] explains, typically adopted solutions, like applying hierarchical or collective forms of responsibility, are wanting, and personal responsibility approaches, based on some weak causal connection between individuals and events, 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 for system engineering, we actually constrain the ways in which software is designed and developed.

3 Engineering MAO with Accountability/Responsibility

Since the proposal is set into the JaCaMo framework [10] (whose organization model is briefly introduced below), the coordinated execution of the agents is regulated by obligations, that are issued by the organization. In [2], it is 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. As in that proposal, we denote by $\mathsf{R}(x, q)$ and $\mathsf{A}(x, y, r, u)$ responsibility and accountability relationships, respectively. $\mathsf{R}(x, q)$ expresses an expectation on any agent playing role x on pursuing condition q (x is in position of being considered to control 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 an agreed standard which brings about expectations inside the organization. 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, besides introducing JaCaMo organizational model, we discuss a programming pattern for accountable agents, that is, for programming 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. We will also describe a full implementation of JaCaMo with accountabilities.

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 [25], involving Jason [11] autonomous agents, working in a shared, artifact-based environment, programmed in CArtAgO [33]. 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 *body* should be taken.

$\mathcal{M}$OISE includes an organization modeling language and an organization management infrastructure [24]. 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 [23]. 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 of the obligation becomes *inactive*.

Accountability/Responsibility Specifications in the JaCaMo Organisation Model

To specify the execution conditions that are object of accountability and responsibility, we use the event-based linear logic called *precedence logic* [34]. 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 [28, 34], allows tracking 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 [34, 28]. 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:

$0/e \doteq 0$	$\top/e \doteq \top$
$(r \wedge u)/e \doteq ((r/e) \wedge (u/e))$	$(r \vee u)/e \doteq ((r/e) \vee (u/e))$
$(e \cdot r)/e \doteq r$, if $e \notin \Gamma_r$	$(e' \cdot r)/e \doteq 0$, if $e \in \Gamma_r$
$r/e \doteq r$, if $e \notin \Gamma_r$	$(\overline{e} \cdot r)/e \doteq 0$

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

In general, 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 [2]. 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. Thus, fitting also provides a guide for developing agents that are *accountable by design*, because it expresses (1) what each agent is engaged to achieve, by fulfilling its responsibilities, and (2), through accountability, how this achievement is related to that process which is the goal of the organization.

In other words, $\mathbf{R} \rightsquigarrow \mathbb{A}$ 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.

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 MAO. Leveraging these concepts for developing agents provides interesting advantages from a software engineering point of view.

As a tool for realizing the accountability fitting that specifies an organization, we are about to introduce a *programming pattern* that allows 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.*

Thanks to a proper programming pattern, for all agents playing role x, the fitting projection over role x can be mapped into a number of Jason plans that will be part of the actual agent program. We provide such a pattern in a way that suits JaCaMo (i.e. the setting of this work) by exploiting the obligations implied by accountabilities and responsibilities [21]. The pattern is expressed in AgentSpeak(ER) [32] because it allows encapsulating a set of plans into a same context that, in our case, depends on x player being accountable towards another agent y about some condition q, and will be adopted until for some reason the drop its responsibility inside the organization (e.g. the agent leaves the organization). More in details, AgentSpeak(ER) extends Jason by introducing two types of plans: *g-plans* and *e-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. E-plans are defined in the scope of a g-plan, and 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 certain events as events it should tackle, by means of some behavior of its own, thanks to the part of its identity that is provided by the organizational role it plays. The agent will also be aware of its social position both (1) by knowing which other agents 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 1st Level Support Agent could decide to handle a task directly or ask to 2nd Level Support). 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. Sometimes, the account of an agent that behaved as expected will be evident to the interested agents from the way in which it operated in the environment [20]. In this case, the obligation to give an account for the satisfaction of an obligation is implicitly resolved by satisfying the very same obligation, and, there is no need to explicitly capture the obligation to provide an account. 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.

Wrong-doing e-plan. The second pattern allows agents to provide accounts also in the case they did 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 Shaping Business Processes as Accountable Agents in JaCaMo

In JaCaMo [10] the state of an organization is encoded in terms of group instances, that map agents to the roles they play, and scheme instances, that allow tracing which goals have been achieved and which are ready for pursue. By exploiting these instances, the organization issues proper obligations to the agents. Due to the declarative nature of scheme instances, agents can autonomously decide how to satisfy their obligations. They are, however, held to notify the organization about the completion of their tasks by means the special directive *goalAchieved*.

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

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

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

1. For each process, a corresponding *manager* role in the organization is defined. The agent(s) playing this role will have to decide on the alternative branches to choose in the process execution;
2. For all the activities in a process, suitable *worker* roles in the organization are defined. These roles will be played by the agents in charge of executing the activities;
3. For each process:
   - A group collecting the manager and all the workers involved in the process is defined;
   - A social scheme is created to organize the activities as a goal decomposition tree[3]. Corresponding missions are defined, to be assigned to roles of the group in charge of the process by defining corresponding norms;
4. For each set of activities to be executed in sequence, the corresponding goals are added to the social scheme by means of the "sequence" operator;

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

5. For each set of activities to be executed without strict ordering, the corresponding goals are added to the social scheme by means of the "parallel" operator;
6. If a choice is present inside a process, a corresponding goal is added to the social scheme by means of the "choice" operator. Each subgoal represents a possible course of action (alternative branch). Every alternative in the choice should include a goal, encoding the chosen path to be assigned to the process manager. Depending on which goal will be achieved by the manager, the execution will follow a branch or another;
7. If a process sends a message that makes another process start, the message should be sent to the process manager, which, as a consequence, will instantiate the social scheme corresponding to the process;
8. If a process includes waiting for a message from another process to proceed, a corresponding goal is added to the social scheme and assigend to the manager; such goal is to be set as achieved only after the message is received. The introduction of this goal is necessary to ensure the coordination, and synchronization during the execution, of the social schemes corresponding to the two processes.

Example 1 (Incident Management as a JaCaMo Organization). Let us now explain how the Incident Management scenario can be mapped into a JaCaMo organization by applying these steps. For the sake of simplicity, we will just consider the Key Account Manager process. First, we introduce a manager role *am* and, for each activity in the process, we define a corresponding worker. aw_1, for instance, will be in charge of *get-description*, aw_2 will be in charge of *explain-solution*, and so on. As a further step we define a Key Account group collecting *am* and all of its workers. At the beginning of the execution, the customer agent *c* will send a message to *am* reporting the problem. As a consequence the agent playing role *am* will instantiate a scheme that will be assigned to this group and will encompass the overall Key Account Manager process. The root goal of such scheme will be assigned to *am* that, in order to satisfy it, will have to manage the successful execution of the social scheme.

Figure 2 shows the scheme available for instantiation to *am* that represents the possible courses of actions during the execution of the Key Account Manager process. Subgoals in a sequence are anticipated by a number denoting the position of the goal in the sequence. Goals including a choice are underlined. The picture also shows how goals are grouped together into missions and which agents such missions are assigned to (through norms). The scheme is to be instantiated as soon as a problem to be solved is reported. Indeed, we map the scheme instantiation to *report-problem*. As a consequence, the obligation to achieve *get-description* will be issued to the corresponding worker aw_1. After the successful achievement of *get-description*, two obligations under a choice will be issued towards *am*: the former related to the *can-handle* goal and the latter to *cannot-handle*. Depending on the result of the previous activity (i.e., whether the problem requires further support or can be handled at that level), *am* will decide to achieve either one of the two goals, the choice made by *am* thus constrains the subsequent obligations that will be generated. In the former case, the normative system will simply issue the obligation to *explain-solution*, while in the latter it will issue the obligation to ask support to the first level. In the second case, after a request for support has been made, again two options are available to the manager. If an answer is received

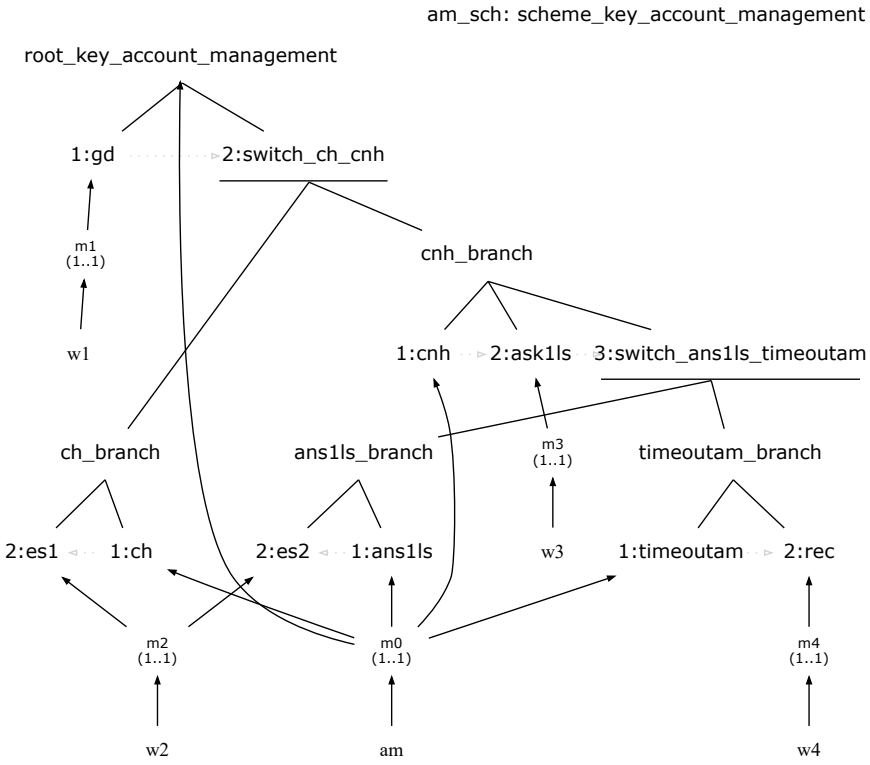

Fig. 2: Social scheme realizing the Key Account Manager process.

from the first level support, the solution has to be explained; in this case *am* will achieve *answer-from-fls*. On the contrary, if a feedback is not received after one day, causing a timeout in the BPMN diagram, an invitation to recall will be sent to the customer. ■

4.1 Adding Accountabilities and Responsibilities

The approach described above is applied to the three remaining processes in example, whose translation in JaCaMo is not discussed here for the sake of simplicity. The organization specification we obtain can, then, be enriched with an accountability specification that allows us to capture BPMN exceptions.

Example 2 (Incident Management with Accountabilities). Figure 3 shows an excerpt of an accountability specification $\mathbf{A}_{incident}$ for the incident management scenario. Accountabilities a_1-a_6 concern the Key Account Manager Process. For instance, accountability a_1 states that *am* is accountable towards c, the customer, for the problem management after a problem is reported [4], and legitimates c, as a-taker, to ask for an account

[4] Event *problem-management* corresponds to the achievement of $root\_key\_account\_management$, the root goal of the social scheme in Figure 2.

$\boldsymbol{a_1}$: A(*am*, *c*, *report-problem*, *problem-management*)
$\boldsymbol{a_2}$: A(aw_1, *am*, *report-problem*, *get-description*)
$\boldsymbol{a_3}$: A(aw_2, *am*, *report-problem* · *get-description* · *can-handle*, *explain-solution*)
$\boldsymbol{a_4}$: A(aw_3, *am*, *report-problem* · *get-description* · *cannot-handle*, *ask-fl-support*)
$\boldsymbol{a_5}$: A(aw_2, *am*, *report-problem* · *get-description* · *cannot-handle* · *ask-fl-support* · *answer-from-fls*, *explain-solution*)
$\boldsymbol{a_6}$: A(aw_4, *am*, *report-problem* · *get-description* · *cannot-handle* · *ask-fl-support* · *1-day*, *invite-recall*)
$\boldsymbol{a_7}$: A(*flm*, *am*, *ask-fl-support*, *provide-feedback-am*$_{fls}$)
...

$\boldsymbol{r_1}$: R(*am*, *problem-management*) $\boldsymbol{r_2}$: R(aw_1, *get-description*) $\boldsymbol{r_3}$: R(aw_2, *explain-solution*)
...

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

about the management of the problem. The a-giver *am* is, thus, expected to provide such an account when needed and recognizes the legitimacy of such an expectation. Accountability a_2, instead, states that once a problem has been reported (and the manager has started the process), the manager has the right to ask worker aw_1 an account about *get-description* (i.e., the achievement of goal gd in the social scheme).

Accountabilities a_3 and a_4 cover the two alternative paths after *get-description*. a_3 encodes the fact that, if the problem can be handled directly, a worker aw_2 will be in charge of *explain-solution* (the achievement of es1 in the scheme). It's worth noting that we require the same worker to be accountable for that activity also in another context, as stated by a_5. This accountability states that the same worker should account for *explain-solution* if the problem cannot be handled directly, too, but in case support has been requested to the first level support and an answer provided (see es2 in the social scheme). Accountability a_4, conversely, states that another worker aw_3 is accountable for asking further support if the problem cannot be handled. Accountability a_4, finally, is related to the *invite-recall* condition. Should an answer not be received within one day from the first level, a worker aw_4 would be accountable for the *invite-recall* (rec in the social scheme) activity. These accountabilities, together, completely characterize the Key Account Manager process[5]. Accountability a_7, in turn, involves the manager of the first level of support. It states that such agent is accountable towards *am* for providing a feedback, once a request for support has been made. In this case, too, such accountability will be supported by further accountability relationships, not reported here, built upon the First Level Support process' decomposition tree. ■

Accountabilities capture only a part of the organization specification. A business process captures an activity flow, that is, it involves activities that are meant to be executed. It is, thus, necessary to identify those agents which are in position of being capable of carrying out the various activities. This is captured by responsibilities. For instance, in the case at issue, due to $r1$, am is responsible of *problem-management*.

[5] It's worth noting that we do not define any accountability relationship w.r.t. *can-handle*, *cannot-handle*, *answer-from-fls* and *1-day*. Such goals are the ones whom the manager is in charge of and would encompass the manager both as a-giver and a-taker.

In order for the organization to execute the business process so that a full account can be provided, it is necessary to bind accountabilities with responsibility assumptions. This is done by the accountability fitting, that, by exploiting the pattern described in the previous section, enables the implementation of exception handling mechanisms. 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 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:

$$\textit{problem-management}/\textit{report-problem}/\textit{problem-management} \equiv \top$$

We now briefly explain how, by relying on an accountability specification and on the programming patterns described above, it is possible to implement an agent playing role *am* whose behavior is accountable for a given set of relevant events.

Example 3 (Key Account Manager with Exceptions). First of all, an agent playing role *am* has to instantiate the scheme for the Key Account Manager (see Figure 2) process as soon as it receives a request from a customer. This behavior is realized through the following plan:

```
+problem
   : group(g1,key_account_management_group,GrArtId)
  <- createScheme(am_sch, scheme_key_account_management, SchArtId);
     addScheme(am_sch)[artifact_id(GrArtId)].
```

Let us now show how to apply the programming pattern base on the accountability specification discussed above. For instance, let us consider the fitting $r_1 \rightsquigarrow a_1$: the agent being developed must be in control of responsibility r_1 (i.e., *problem-management*), and assume an accountable behavior towards the customer for *problem-management*, when condition *report-problem* holds. We recall that we map *problem-management* to the achievement of the root goal of the scheme in Figure 2, and *report-problem* to the instantiation of such scheme. Following the pattern described in the previous section, the programmer should define in the agent body a g-plan containing *well-doing* and *wrong-doing* e-plans. However, the actual framework we use for implementation is JaCaMo, where Jason still implements AgentSpeak(L) rather than AgentSpeak(ER), and hence it is not possible to define nested e-plans. The *well-doing* and *wrong-doing* e-plans are therefore rendered by means of two plain Jason plans. The fitting involving $r_1 \rightsquigarrow a_1$ is implemented by the following plans:

```
account_to(ATaker,root_key_account_management) :- play(ATaker,customer,g1).

+obligation(Ag,_,What,_)
   : .my_name(Ag) &
     done(am_sch,root_key_account_management,Ag)=What &
     scheme(am_sch,_,SchId)
  <- goalAchieved(root_key_account_management)[artifact_id(SchId)].

+oblUnfulfilled(O)
   : .my_name(Ag) &  obligation(Ag,_,What,_) = O &
     done(am_sch,root_key_account_management,Ag)=What &
     scheme(am_sch,_,SchId) &
     account_to(ATaker,root_key_account_management)
  <- .send(ATaker, tell, reason_for_failure).
```

The first plan, in particular, realizes the *well-doing* part of the pattern. Recalling Definition 2, the plan is triggered as soon as an obligation concerning the scheme root is issued (Line 3). The obligation object (*What*) is the satisfaction of the organizational goal `root_key_account_management` –corresponding to the *problem-management* condition 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_1. In JaCaMo we represent this condition (*report-problem*) in terms of scheme being instantiated (see Line 6). Considering the fitting-adherence condition, we have that r is *report-problem*, and u is *problem-management*. Thus $q \equiv u/r$ is just *problem-management*. The plan for *well-doing* needs to include some actions that amount to such an event. In this setting we consider the event to be occurred as soon as the corresponding organizational goal is set as achieved. This is trivially true in the example (see Line 7). The second plan, at Line 9, instead, deals with the *wrong-doing* part of the pattern. Should, for any reason, the obligation be unfulfilled, the agent, by virtue of its accountability, must provide a motivation about the unsatisfaction of the obligation. In this case, a proper message encoding the explanation for the failure is sent to the account-taker agent (see Lines 14). Note that the agent could be equipped with multiple plans covering the wrong-doing part to take care of several causes of failures each with its own specific explanation to be sent to a precise a-taker. These two plans ensure that, no matter how the scheme execution evolves, *c* will always receive an account from *am*, consisting either of a notification about the achievement of *problem-management*, or of an explanation for the failure. ■

The implementation of an agent playing the *am* role must also take into account that the agent is the manager of the Key Account Manager process. Therefore the agent should also be provided with plans for *controlling the execution flow*, that is, for deciding which goals amount to choices it needs to satisfy.

Example 4 (Controlling Alternatives). In our case there are two alternative goals ch and cnh, whose achievement corresponds to *can-handle* and *cannot-handle*), respectively. Specifically, after goal *get-description* gets accomplished, the organizational infrastructure will issue two obligations at the same time, directed to *am*, one for ch and one for cnh. The agent will then decide to achieve either one of the two goals depending on data that are not captured by the functional decomposition, but that may be accessible to the agent by means of artifacts. The two plans below realize this behavior:

```
1  +obligation(Ag,_,What,_)
2     : .my_name(Ag) &
3       done(am_sch,ch,Ag)=What &
4       description(easy_problem) &
5       scheme(am_sch,_,SchId)
6    <- goalAchieved(ch)[artifact_id(SchId)].
7
8  +obligation(Ag,_,What,_)
9     : .my_name(Ag) &
10      done(am_sch,cnh,Ag)=What &
11      description(hard_problem) &
12      scheme(am_sch,_,SchId)
13   <- goalAchieved(cnh)[artifact_id(SchId)].
```

If the problem is directly solvable, the agent will fulfill the first obligation, otherwise it will satisfy the second one. The choice of which obligation is actually fulfilled obviously affects the subsequent activities. In the second case, in fact, a new sub-process needs to be activated, that is, a new functional scheme needs to be instantiated. ■

Worker agents for the Key Account Manager process are developed in a similar way, following the pattern. For instance, let's consider the agent playing the aw_1 role. The agent is involved in $r_2 \rightsquigarrow a_2$. The well-doing and wrong-doing plans are defined as follows:

```
1  account_to(ATaker,gd) :- play(ATaker,key_account_manager,g1).
2
3  +obligation(Ag,_,What,_)
4     : .my_name(Ag) &
5       done(am_sch,gd,Ag)=What &
6       play(C,customer,g1)
7    <- .send(C,tell,ask_description).
8
9  +description(D)
10    : account_to(AccountTaker,gd) &
11      scheme(am_sch,_,SchId)
12   <- .send(AccountTaker,tell,description(D));
13      goalAchieved(gd)[artifact_id(SchId)].
14
15 +oblUnfulfilled(O)
16    : .my_name(Ag) &  obligation(Ag,_,What,_) = O &
17      done(am_sch,gd,Ag)=What &
18      scheme(am_sch,_,SchId) &
19      not description(D) &
20      account_to(ATaker,root_key_account_management)
21  <- .send(ATaker, tell, no_description_from_customer).
```

The first two plans, together, realize the well-doing part of the pattern. The first plan (Line 3), in particular, is triggered as soon as the obligation to achieve `gd` is issued. As a result, the agent will send a message to the customer asking for a description. To successfully complete the activity and achieve its goal, however, the agent has to wait for an answer. The second plan, at Line 9, is triggered as soon as a description is received from the customer and finally lead to the achievement of the organizational goal (Line 13). Before that, nonetheless, the worker forwards the description to his manager (Line 12), which will then use it to decide how to proceed, as explained above. The plan at Line 15 realizes the wrong-doing part of the pattern. In this case, should the obligation become unfulfilled because of a missing response from the customer (see Line 19), the account sent to the a-taker would be `no_description_from_customer` (see Line 21).

The *am* agent, in turn, being account-taker in several accountability relationships, can include also some plans to handle the accounts provided by its account-givers, both in positive and negative circumstances. Let's consider again a_2. As explained above, in case of failure, aw_1 would send a precise message to *am*. The very same message can be used as triggering event in a recovery plan, as follows:

```
1  no_description_from_customer
2     : play(C,customer,g1) &
3       scheme(am_sch,_,SchId)
4    <- .send(C,tell,please_answer);
5       resetGoal(gd)[artifact_id(SchId)].
```

A possible way to deal with the failure could be to send a further message to the customer and reset the failed goal. In general considerations related to how to handle a failure are strictly domain dependent.

Notably, considering the accountability specification as a requirement, the actual implementation of the system results robust. The accountable *am* for instance, to satisfy the requirement of being accountable, must be capable, on the one side, of capturing exceptions from other agents, and on the other side, of providing an account to its a-taker (i.e., the customer).

5 Conclusions

We have discussed how the agent technology can be exploited for the implementation of business processes modeled as BPMN schema. To be effective, however, agent system needs to take care of some peculiarities of business processes, as for instance, the handling of exceptional events, that possibly interrupt the execution of a process. Approaches for modeling exceptions in a multi-agent setting have been proposed (see, e.g., [27, 35, 31]), but these proposal fall short in addressing the typical properties of MAS, such as autonomy, openness, heterogeneity, and encapsulation.

In this paper we have coped with BPMN exceptions by relying on the notions of accountability and responsibility as elements for the specification of an agent organization. In addition, we have proposed a programming pattern for developing agents that adhere to such a specification. The pattern, when applied systematically, brings 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 (see [1]). In fact, while a responsibility distribution is a coverage of the functional decomposition, an accountability specification conveys how the agents contribute to the process. Hence, the accountabilities provide the programmer with a behavioral specification agents must satisfy.

Our approach is specular to [39], whose aim 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 inferred plan. 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., [22]), 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. Accountabilities can fill in this gap: when an obligation is not satisfied, it is reasonable to report the exception to the a-taker. This is achieved quite naturally with the *Wrong-Doing Pattern*, that allows an agent to provide an account

for an unsatisfied obligation. Interestingly, the choice "on-the-fly" of which branch to follow, performed by the manager, 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" [26]. The integration of this type of autonomy into an organizational model (i.e., $\mathcal{M}$OISE) discussed in [26] opens interesting perspectives in the modeling of BPMN processes for our accountable agents.

Commitment-based protocols (e.g., [40]), 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. 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)
19. 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
20. Garfinkel, H.: Studies in ethnomethodology. Prentice-Hall Inc., Englewood Cliffs, New Jersey (1967)
21. Grant, R.W., Keohane, R.O.: Accountability and Abuses of Power in World Politics. The American Political Science Review **99**(1) (2005)
22. Haller, P., Sommers, F.: Actors in Scala - concurrent programming for the multi-core era. Artima (2011)
23. 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)
24. 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)
25. 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)

26. 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)
27. 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)
28. 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)
29. Nissenbaum, H.: Accountability in a computerized society. Science and Engineering Ethics **2**(1), 25–42 (1996)
30. Object Management Group: Bpmn specification - business process model and notation (2018), `http://www.bpmn.org/`, online, accessed 08/11/2018
31. 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)
32. 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)
33. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment Programming in CArtAgO, pp. 259–288. Springer US, Boston, MA (2009)
34. 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)
35. 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)
36. Thomson, J.J.: Remarks on causation and liability. Philosophy and Public Affairs **13**(2), 101–133 (1984)
37. Vincent, N.A.: Moral Responsibility, Library of Ethics and Applied Philosophy, vol. 27, chap. A Structured Taxonomy of Responsibility Concepts. Springer (2011)
38. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer (2007)
39. 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
40. Yolum, P., Singh, M.P.: Commitment Machines. In: Intelligent Agents VIII, 8th Int. WS, ATAL 2001. LNCS, vol. 2333, pp. 235–247. Springer (2002)