Autonomous Agents and Multi-Agent Systems (2023) 37:7
https://doi.org/10.1007/s10458-022-09590-6

Accountability in multi-agent organizations: from conceptual design to agent programming

Matteo Baldoni[1] · **Cristina Baroglio**[1] · **Roberto Micalizio**[1] · **Stefano Tedeschi**[1]

Accepted: 1 November 2022 / Published online: 28 November 2022

Abstract
This work proposes a notion of accountability for multi-agent systems, that supports the development of robust distributed systems. Accountability is grounded on responsibility, and encompasses both a normative dimension, and a structural dimension. For realizing robust distributed systems, conceived as agent systems or organizations, it is necessary to keep a right level of situational awareness, through the introduction of the means for gathering and propagating accounts, upon which actions can be taken. This paper presents a formalization of accountability, including the accountability lifecycle, for the design of robust agent organizations. Particular attention is given to the interplay of accountability and goals, by describing typical patterns in which accountability affects the state of an agent's goals and vice versa. We illustrate the practical aspects of the proposal by means of JaCaMo (Boissier et al. Sci Comput Program 78(6):747–761, 2013.https://doi.org/10.1016/j.scico.2011.10.004).

Keywords Accountability · Responsibility · Agent organizations · Engineering and programming MAS · Robustness · JaCaMo

1 Introduction

Agent organizations (e.g., [2]) are a well-known abstraction for conceptualizing and developing distributed systems. The organization metaphor is, in fact, a useful mechanism for modularizing code spread over different software components, that are opaque and independent of each other. Agent organizations show the same kind of structure and of

✉ Matteo Baldoni
matteo.baldoni@unito.it

Cristina Baroglio
cristina.baroglio@unito.it

Roberto Micalizio
roberto.micalizio@unito.it

Stefano Tedeschi
stefano.tedeschi@unito.it

[1] Dipartimento di Informatica, Università degli Studi di Torino, Corso Svizzera 185, 10149 Torino, Italy

advantage that sociologist Dave Elder-Vass explains for human organizations: an organization provides a structure of constraints that allow a system consisting of many parts to act as a whole, with the aim of achieving goals that otherwise would not be achievable (or not as easily) [29].

Many approaches to multiagent systems and organizations, e.g. [15, 19, 25, 39], provide the means to design and realize the correct, expected behavior of the system, capturing exactly what agents should do to contribute to the achievement of the organizational goal. For instance, many approaches rely on *norms* (rules, protocols, etc.) to define what is expected of each agent and which sanction is applied when an agent does not comply. Sanctions are intended as deterrents to prevent norm violation, i.e., to keep the execution oriented towards the achievement of the organizational goal.

The problem is that when the system faces an abnormal situation (i.e., a *perturbation*) and some agent *fails* to achieve a goal, sanctions are of little utility, if any [8, 11, 22]. In this case, in fact, the agent may have earnestly tried its best to do what expected, but something which is not under its control hindered the achievement. What is missing in the picture is some support for allowing agents to provide an *account* on what happened, propagating it through an appropriately devised structure inside the organization, for reaching those agents that are equipped with the means for coping with it. Such a tool aims at making the organization more *robust*, that is, capable to keep an acceptable behavior in spite of unforeseen, abnormal, or stressful conditions. We see *robustness* [1] as a crucial property for scaling from agents to agent organizations, and make a proposal that is based on the concepts of *accountability* and *responsibility*.

Accountability is extremely important in the human world. The kind of accountability we refer to is well-described in a report by the United Nations Development Programme (UNDP) [30]. UNDP's accountability framework describes organization-wide processes for monitoring, analyzing, and improving performance in all aspects of the organization. The framework gives managers the means to address recurring and systemic issues, and to incorporate lessons learned into future activities. Inside the framework, accountability is supported, among other things, by formally documented functions, responsibilities, authority, management expectations, policies, processes and instruments for improving performance.

Contribution. This paper proposes a formalization of accountability in multiagent organizations (MAOs). This supports the development of robust distributed systems by enabling the specification of a proper treatment of possible perturbations. In our approach, accountability is grounded on the notion of responsibility, and is introduced along both its *normative* and its *structural* dimensions, to express who is expected to provide an account to whom (and under which circumstances), and to specify further requirements on how responsibilities should be distributed through the agents in order to produce authoritative accounts. The proposal is independent of the specific organizational model, nevertheless, we illustrate the practical use of accountability by exploiting JaCaMo [16]. This paper extends [13] in several ways. In first lieu, the conceptual model for robust organizations is refined. Then, we introduce the lifecycle of accountability, and we show the interplay of accountability and goals by describing six rules that amount to typical patterns, in which accountability affects the state of an agent's goals and vice versa. For each rule we report a corresponding programming schema for the extension of JaCaMo with accountability. This contribution is completed by the discussion of three scenarios where the rules (and schemas) are used for tackling specific problems.

2 Realizing accountable MAOs

The notion of accountability has recently gained the attention of many authors (see e.g., [3, 7, 22, 24]), who see a powerful software engineering tool in it. We agree and add that accountability is fundamental to design and realize robust agent systems. The authors of [1] say on robustness: "*A [property] of a [system] is robust if it is [invariant] with respect to a [set of perturbations].*" Brackets are original and emphasize that a formal treatment of robustness requires a proper system specification. In other terms, robustness is primarily a matter of *good design*, which in turn demands for proper engineering tools. We see in accountability such a tool.

Building upon sociology literature [28, 32, 47], in agent organizations (e.g., [3, 24]), accountability is a mutually accepted social relationship between two parties such that (1) one of the parties ("account taker" or *a-taker*) can legitimately demand, under certain conditions, an account about a process of interest, and (2) the other ("account giver" or *a-giver*) is legitimately required to provide the account. In accountability, we recognize two dimensions [10, 12, 13]:

- The *normative* dimension creates mutual expectations on the behavior of the involved agents; it captures the *legitimacy*, for the account taker, of asking (and the *availability* of the account giver to provide) an account (the standing of the account taker to demand an account).
- The *structural* dimension concerns the *capability* to produce an account; for being held to account about a process, an agent must exert control over the same process and must have proper awareness of the situation it accounts for, possibly by relying on other agents.

In general, agents are allowed to provide accounts for tasks that have been carried out by others, or that involve the intervention of other agents; in order to do so, they must be in position of receiving accounts by the other involved agents. In order for accounts to be reliable, the recursive structure of accountability should allow accounts to always be traced back to the agents that are responsible for the involved tasks. In other terms, recursion closes on some responsibility. This explains why accountability cannot be reduced to the right/obligation involved in the relationship (normative dimension). In [12] we introduced a conceptual model that defines organizational accountability by following this line.

Consequently, accountability is an emergent property of the specification of an organization (see [29] about emergent organizational powers), that shows a certain structure. Taking as references the models of organizations and institutions, discussed in [16, 20, 26, 27, 31, 39, 54], we propose a model of organization, see Fig. 1, where white boxes represent the common concepts of organizations, and yellow boxes introduce concepts and relationships that capture the normative aspects bounding the roles (played by agents) to accounting and treatment tasks.

However, in order for accountability to emerge as a property of the system as a whole, it is also necessary to verify that the structural dimension is adequately captured (see Sect. 3), as well as, it is necessary to capture the correct interplay between agent goals and accountability agreements specified at organizational level (see Sect. 5.2). When this happens, proviso the specified treatment is effective, the resulting accountability-based multi-agent system will be robust against given perturbations.

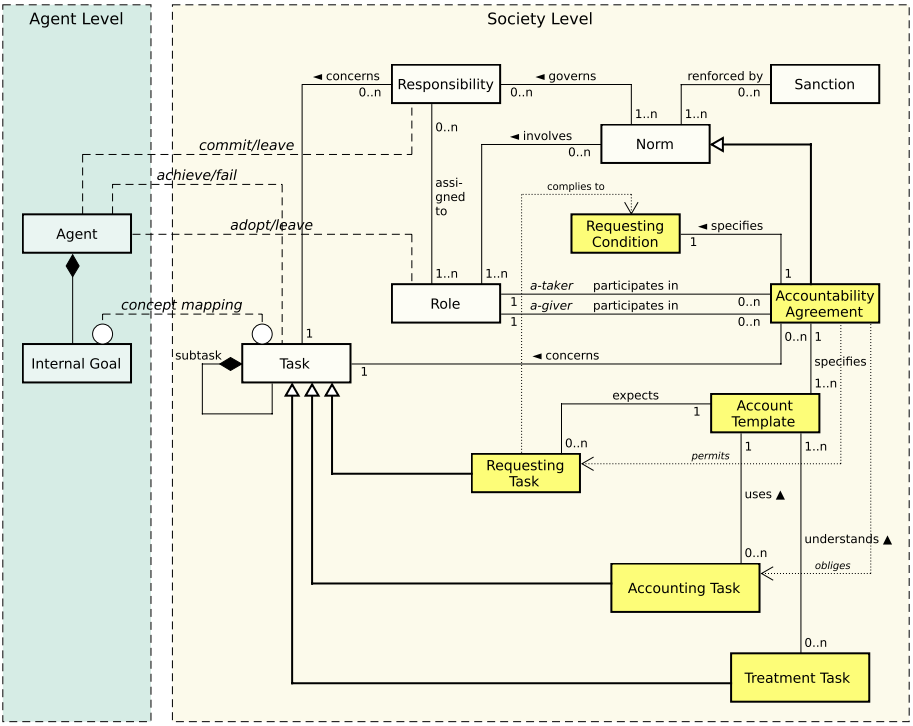

Fig. 1 The proposed model of robust multiagent organizations

As pointed out in the literature, especially [17], organizations allow structuring the activities performed by the agents, and to coordinate them. The society is often shaped by a set of *norms*, that create social expectations through, e.g., commitments, authorizations, prohibitions [49]. Responsibilities are seen as duties that the agents, who accepted to play some `Roles` within the organization, are aware of and have accepted, while obligations are seen as a mechanism for telling agents when and how to discharge their responsibilities, by accomplishing their tasks.

`Norms` absolve this necessity by yielding obligations about the tasks that agents are held to fulfill; they are, therefore, used to describe the expected behavior of agents in terms of their responsibilities, rights and duties [41]. Many methodologies for agent organizations (e.g., [26, 39, 54]) hinge on the concepts of `Task` and of `Responsibility` concerning some `Task`. So, in Gaia [54], the functionality of a role is defined by its responsibilities. The OperA framework [26] is able to define the global aims of an organization (tasks), and the objectives and responsibilities of its participants. A similar idea recurs in other frameworks, such as OMNI [27] and MOISE [39], where a functional decomposition describes how a complex goal (task) can be achieved in a distributed way. Agents joining the organization are expected to contribute by achieving subgoals of such a decomposition, whenever obligations are triggered by the organization toward them. In MOISE, agents are held to explicitly commit to missions (i.e., subsets of goals), this act implies an assumption of responsibility of the agents toward their missions and, hence, the acceptance of the related obligations that will be issued by the organization. `Norms` are reinforced by

Sanctions, and involve Roles, while Responsibilitys are assigned to Roles. An Agent can adopt/leave a Role, commit to/leave a Responsibility, and can achieve/fail a Task by internalizing it as a goal of its own. We assume agents to be autonomous, thus, we require them to explicitly *commit to their responsibilities*. The agent autonomy is preserved, since agents can decide not to satisfy an obligation despite being possibly sanctioned, or they can deliberate how to act to trigger certain obligations. Coming to accountability, the model captures its normative dimension by the concept of Accountability Agreement, the structural dimension, instead, is formally defined in terms of *A-structures* (Sect. 3.1). We use the term *accountability* to encompass both dimensions. Accountability Agreement specializes Norm. An Accountability Agreement between two parties specifies the a-taker and a-giver, the Requesting Condition, i.e., the condition under which a request is allowed, and the Account Template (possibly many), that captures the expected structure of the account. The association "concern" between Accountability Agreement and Task captures the object of the account. That is, the a-giver is expected to produce an account that is relevant for the task indicated via this association, when the a-taker legitimately asks for an account concerning it. Account Templates are crucial because they allow a-taker and a-giver to tune their behavior by specifying the kind of Account one needs and expects from the other. In other words, a Requesting Task expects some Account Template to be followed. Conversely, an Accounting Task uses some Account Template to produce some Account. The characteristic of Requesting Tasks and Accounting Tasks, with respect to plain Tasks, is their relationship with an Account Template.

Robustness is obtained by making the system tolerant to perturbations for which an adequate treatment is known. In presence of perturbations, asking for and obtaining accounts provides the ground for applying an adequate treatment. In the model, Account Templates are associated to Treatment Tasks (a Treatment Task understands some Account Template), and agents playing Roles take on responsibility concerning Treatment Tasks. From a normative perspective, the a-taker is permitted to perform a Requesting Task in order to ask for an account. The a-giver may be obliged to perform an Accounting Task to produce an account. This is captured by means of the twofold association between Accountability Agreement and Agent – in one case, an agent plays the role of *a-taker*; in the other, of *a-giver*.

3 Formalizing the dimensions of accountability

Let us now explain how the normative and the structural dimensions of accountability are formalized in our proposal. An accountability agreement is formally defined as follows.

Definition 1 *(Accountability Agreement)* An accountability agreement, denoted as $\mathbb{AA}(x, y, r, u)$, is a structure where x and y are agents playing roles in the organization; r determines when an account request is legitimate, and u is the object of the accountability agreement.

When r has occurred, two conditions should be implied: (1) y has the claim-right to ask x for an account about u; and (2) x is actually in condition to provide substantive and authoritative accounts about u. This happens when x can retrieve the necessary contextual information about u, either because the agent is directly responsible for the execution of u,

or because it can rely on other agents to gather feedback on the execution of (all the parts of) u. While the first condition is implied by the normative dimension of the accountability, the second condition is only achieved by imposing an adequate structure to the set of defined accountability agreements. Note that an accountability agreement does not imply that x will perform u, but only that x can produce an account about u, possibly by relying on accounts provided by other agents.

We adopt *precedence logic* [48], for expressing r and u, but with different purposes. On the one side, we exploit residuation to determine when a formula is satisfied by a sequence of events occurring in the system. On the other side, precedence logic is used to express how a complex `Task` is decomposed into a workflow of atomic tasks. For the sake of generality of our formalization, an account request about u amounts to asking an account about any of its sub-tasks. Namely, the a-giver will provide as much information as it can about the whole task u (e.g., which atomic tasks have performed successfully, so far.) In the rest of the paper we will denote as $\widetilde{u}$ the account (i.e., a piece of information) that x provides to y about u.

The logic has three primary operators: '$\vee$' (choice), '$\wedge$' (interleaving), and '$\cdot$' (ordering). *Ordering* allows constraining the order with which two events must occur, e.g., $a \cdot b$ means that a must occur before b, but the two events do not need to occur one immediately after the other. Instead, *choice* specifies that at least one of the events should occur, while *interleaving* that all should occur but the order is unimportant. The *residual* of a workflow u with respect to an event e, denoted as u/e, is the remainder workflow that would be left over when e occurs, and whose satisfaction would guarantee the satisfaction of the workflow u. Residual can be calculated by means of a set of rewrite rules. The following equations are due to [48]. Here, u is a workflow, e is an event or $\top$, and $\overline{e}$, the complement of e, is also an event. Initially, neither e nor $\overline{e}$ hold. On any run, either e or $\overline{e}$ may occur but not both. Note that we assume that events are nonrepeating. In practice, we can assume that timestamps differentiate multiple instances of the same event. 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 $\Gamma_{e \cdot f} = \{e, \overline{e}, f, \overline{f}\}$.

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

Where symbols 0 and $\top$ behave as the Boolean constants `false` and `true`, respectively. An event e is *relevant* to a workflow u if that event is involved in u, i.e. $u/e \not\equiv u$ [6]. We use the expression $r/(e_1, \ldots, e_n)$ as a shortcut for $((r/e_1)/\ldots)/e_n$. Finally, let $w = (e_1, \ldots, e_k)$ and $z = (e_{k+1}, \ldots, e_n)$ be two sequences of events, wz is their concatenation $(e_1, \cdots, e_k, e_{k+1}, \ldots, e_n)$.

Definition 2 *(Inclusion)* We denote by $u[r]$ the fact that the workflow u contains as a subformula the workflow r.

For example, if $u = a \wedge (b \cdot c)$, we can write $u[b \cdot c]$ because of $b \cdot c$ is contained in u. Of course, we have that $\forall u : u[u]$; i.e., a workflow is trivially a sub-workflow of itself.

Definition 3 *(Entailment)* we denote by $u \to u'$ the fact that for any $(e_1, \ldots, e_n)$ such that $u/(e_1, \ldots, e_n) = \top$ we also have that $u'/(e_1, \ldots, e_n) = \top$.

Example 1 *(Bakery)* Let us consider a bakery in which the process of bread selling involves the following steps: first the dough is kneaded and, at the same time, the oven is set up for baking. Once the dough and the oven are ready, the bread is actually baked and finally it is sold to customers.

This workflow can be modeled in precedence logic as: $(\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell}$.

Let us now consider agreement $\mathbb{AA}(harold, customer, \mathsf{order}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell})$: *customer* has the right to ask *harold* (i.e., the bakery owner), about the whole baking process provided that she has placed an order. It must be noticed, however, that such an agreement alone is not an accountability. *harold*, in fact, may not be directly involved in the production process, which could be carried out by his employees. To provide a sound account on the process, *harold* needs to rely on the accounts of his employees. This can be achieved by organizing the agreements into a tree-shaped structured as discussed in the following section and exemplified in Example 3.

Given $\mathbb{AA}(x, y, r, u)$, we define its progression against the occurrence of an event e as $\mathbb{AA}(x, y, r, u)/e = \mathbb{AA}(x, y, r/e, u)$. When r/e progresses to $\top$, y is legitimated to ask x for an account, and x is required upon request, to provide y with an account of u.

3.1 Accountable workflows

As mentioned above, accountability is characterized by a structural dimension that is grounded on the responsibilities taken up by the agents. In the following, we denote as $\mathsf{R}(x, u)$ that x has the responsibility for task u. In agreement with Hart's view of role responsibilities [38, p. 212], $\mathsf{R}(x, u)$ yields an expectation over the agent x (that is, x is expected to execute u when needed), because of some role x plays in the organization. Like accountability agreements, responsibilities are preserved at run-time, that is, they are not affected by progression.

Definition 4 *(Grounded Workflow)* Given a workflow u, we say that a set of responsibilities $\mathbf{R}_u$, that contains a responsibility $\mathsf{R}(x, u')$ for each atomic task $u[u']$ and any agent x, is a distribution of responsibilities. In this case, we say that u is grounded on $\mathbf{R}_u$.

Relying on the notion of grounded workflow, we can formally define accountability, and in particular its structural dimension, by means of *accountability structures* (A-structure) and two operations for merging them.

Definition 5 *(A-structure)* An *A-structure* is a pair $\langle A, T \rangle$, where:

- A is a set of accountability agreements;
- T is either a set of responsibilities (in this case we call the A-structure an *A-leaf*) or a set of A-structures.

Intuitively, an A-structure is the backbone upon which the structural dimension of accountability can be achieved. Specifically, the definition of accountability (see Definition 8) will characterize A-structures so that A-leaves will represent one-step accounts: situations where the agent providing an account about a workflow is also responsible for the very same workflow, and this guarantees the soundness of accounts. Soundness, however, is guaranteed also when agents account for workflows of which they are not directly

responsible, see below. As shown in the following definitions, the two operators A-union and A-join allow building A-structures consistently with the operators of the precedence logic.

Definition 6 *(A-union)* The *A-union* between two A-structures, denoted by $\langle A_1, T_1\rangle \oplus \langle A_2, T_2\rangle$, is either:

- $\langle A_1 \cup A_2, T_1 \cup T_2\rangle$, if both the A-structures are A-leaves, or none of them is an A-leaf;
- $\langle A_1 \cup A_2, T_1 \cup \{\langle A_2, T_2\rangle\}\rangle$, if $\langle A_2, T_2\rangle$ is an A-leaf and $\langle A_1, T_1\rangle$ is not;
- $\langle A_1 \cup A_2, T_2 \cup \{\langle A_1, T_1\rangle\}\rangle$, if $\langle A_1, T_1\rangle$ is an A-leaf and $\langle A_2, T_2\rangle$ is not.

Definition 7 *(A-join)* The *A-join* between two A-structures, denoted by $\langle A_1, T_1\rangle \otimes \langle A_2, T_2\rangle$, is recursively defined as follows:

- $\langle A_1, T_1\rangle \oplus \langle A_2, T_2\rangle$, if $\langle A_1, T_1\rangle$ and $\langle A_2, T_2\rangle$ are A-leaves;
- $\langle A_1 \cup A_2, \{t_i \otimes t_j, \forall (t_i, t_j) \in T_1 \times T_2\}\rangle$, otherwise.

The following definition of accountability specifies how an A-structure should be defined over a grounded workflow u, whose atomic tasks are each under the responsibility of some agent. This assures that it is always possible to provide sound accounts for any sub-workflow u' of u (i.e., $u[u']$). This is possible by way of the structural dimension recursively encoded in an A-structure $\langle A, T\rangle$: each accountability agreement in A is supported by the sub-A-structures in T, each of which recursively brakes down to a set of responsibilities for the atomic tasks mentioned in the agreement, in accordance with Definition 4. Indeed, the sub-A-structures in T represent alternative workflow runs, that is, alternative scenarios yielding an account for the workflow mentioned in the agreements in A.

Definition 8 *(Accountability)* Let u be a workflow that is grounded on $\mathbf{R}_u$, an accountability $\mathbb{A}(x, y, r, u)$ over $\mathbf{R}_u$ is an A-structure $\langle\{\mathbb{AA}(x, y, r, u)\}, T\rangle$ defined as follow:

- T is $\{\mathsf{R}(x, u)\}$, such that $\mathsf{R}(x, u) \in \mathbf{R}_u$;
- $\mathbb{A}(x, y, r, u) = \mathbb{AA}(x, y, r, u) + \mathbb{A}(z, x, r, u)$;
- $\mathbb{A}(x, y, r, u' \vee u'') = \mathbb{A}(x, y, r, u') \vee \mathbb{A}(x, y, r, u'')$;
- $\mathbb{A}(x, y, r, u' \wedge u'') = \mathbb{A}(x, y, r, u') \wedge \mathbb{A}(x, y, r, u'')$;
- $\mathbb{A}(x, y, r, u' \cdot u'') = \mathbb{A}(x, y, r, u') \cdot \mathbb{A}(x, y, r \cdot u', u'')$.

Where the operations $+$, $\vee$, $\wedge$, and $\cdot$ on accountabilities are defined as follows, supposing $\mathbb{A}(x, y, r, u') = \langle A_{u'}, T_{u'}\rangle$, $\mathbb{A}(x, y, r, u'') = \langle A_{u''}, T_{u''}\rangle$, $\mathbb{A}(z, x, r, u) = \langle A_u, T_u\rangle$:

- $\mathbb{AA}(x, y, r, u) + \mathbb{A}(z, x, r, u) = \langle\{\mathbb{AA}(x, y, r, u)\}, \{\langle A_u, T_u\rangle\}\rangle$;
- $\mathbb{A}(x, y, r, u') \vee \mathbb{A}(x, y, r, u'') = \langle\{\mathbb{AA}(x, y, r, u' \vee u'')\}, \{\langle A_{u'}, T_{u'}\rangle \oplus \langle A_{u''}, T_{u''}\rangle\}\rangle$;
- $\mathbb{A}(x, y, r, u') \wedge \mathbb{A}(x, y, r, u'') = \langle\{\mathbb{AA}(x, y, r, u' \wedge u'')\}, \{\langle A_{u'}, T_{u'}\rangle \otimes \langle A_{u''}, T_{u''}\rangle\}\rangle$;
- $\mathbb{A}(x, y, r, u') \cdot \mathbb{A}(x, y, r \cdot u', u'') = \langle\{\mathbb{AA}(x, y, r, u' \cdot u'')\}, \{\langle A_{u'}, T_{u'}\rangle \otimes \langle A_{u''}, T_{u''}\rangle\}\rangle$.

We use A_u as a shortcut for a singleton set $\{\mathbb{AA}(x, y, r, u)\}$ for some agents x and y, and some condition r and workflow u.

The above definition points out a compositional feature of accountability: a complex workflow can be accounted for by an agent when such an agent can exploit accounts from

others about every portion of the workflow itself. This is possible thanks to the two operations upon A-structures. The union between two A-structures is used to collect alternative ways for accounting a workflow when its main operator is a choice. On the other hand, when the main operator of a workflow is interleaving or ordering, the join between two A-structures is used to combine the ways for accounting for the two sub-workflows.

Example 2 In the bakery example, let us consider the following responsibility distribution $\mathbf{R} = \{\mathsf{R}(mike, \mathsf{knead}), \mathsf{R}(bart, \mathsf{heatOven}), \mathsf{R}(bart, \mathsf{bake}), \mathsf{R}(sheila, \mathsf{sell})\}$. We can define the accountability of *harold*, the bakery owner, towards a possible *customer* over $\mathbf{R}$ as: $\mathbb{A}(harold, customer, \mathsf{order}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell})$. However, since *harold* is not directly responsible for the whole workflow, such an accountability is well founded only if it is grounded on the distribution $\mathbf{R}$ of responsibilities. That is, only when it is possible to recursively demonstrate, by exploiting Definition 8, that every sub-workflow in $(\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell})$ is grounded over $\mathbf{R}$. Figure 2 sketches the structural dimension underpinning accountability $\mathbb{A}(harold, customer, \mathsf{order}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell})$. Indeed, this is a simplified representation of the A-structure supporting the accountability. The A-leaves show that the accountability about atomic tasks is supported by responsibilities in $\mathbf{R}$. Intermediate nodes represent how the accountability of a sub-workflow is gained by combining the accountabilities of its parts. Thus, the structure of accountability ensures that *customer* will be in condition to receive an authoritative account about any part of the workflow, let us say bake, from *harold*. Despite not being directly involved in the task, *harold* can recursively gather an account about it thanks to *bart*'s accountability.

Definition 8 says accountability is a tree by construction, whose nodes have form $\mathbb{A}(x, y, r, u) = \langle \{\mathbb{AA}(x, y, r, u)\}, T_u \rangle$. $\mathbb{AA}(x, y, r, u)$ is an accountability agreement, and T_u is the structure through which a sound account of u can be delivered to x. Thanks to this tree, it is possible to navigate through accountabilities along feedback chains.

Definition 9 *(Feedback chain)* A feedback chain is a sequence of $\langle \mathbb{A}(x_0, y_0, r_0, u_0), \ldots, \mathbb{A}(x_n, y_n, r_n, u_n) \rangle$ such that for each $i = 1, \ldots, n$, we have that $u_{i-1}[u_i]$, $y_i = x_{i-1}$, and $r_i \rightarrow r_{i-1}$.

Grounding the tree over responsibilities (A-leaves) ensures that each agent, which is involved in the structure, has the means for gaining situational awareness of the process it is involved into; consequently, it can provide sound accounts. We formalize this as a property of accountable workflows. Proposition 1 guarantees that, given a workflow u, it is possible to define a proper set of feedback chains, from a-givers to the corresponding a-takers, for every sub-workflow of u. Thanks to responsibility, a-givers are the agents that are competent for each subprocess of u, and hence can provide sound accounts.

Proposition 1 (Accountable workflow) *Let u be a workflow and let $\mathbb{A}(x, y, r, u)$ be an accountability over $\mathbf{R}_u$. There exists a feedback chain $\langle \mathbb{A}(x, y, r, u), \ldots, \mathbb{A}(x', y', r', u') \rangle$ for any workflow u' such that $u[u']$. We call u an accountable workflow through $\mathbb{A}(x, y, r, u)$ over $\mathbf{R}_u$.*

Proof The proof is by induction on the structure of $\mathbb{A}(x, y, r, u)$.

In the base case, u is atomic so $\mathbb{A}(x, y, r, u)$ is $\langle \{\mathbb{AA}(x, y, r, u)\}, \{\mathsf{R}(x, u)\} \rangle$, and the chain is given by $\langle \mathbb{A}(x, y, r, u) \rangle$.

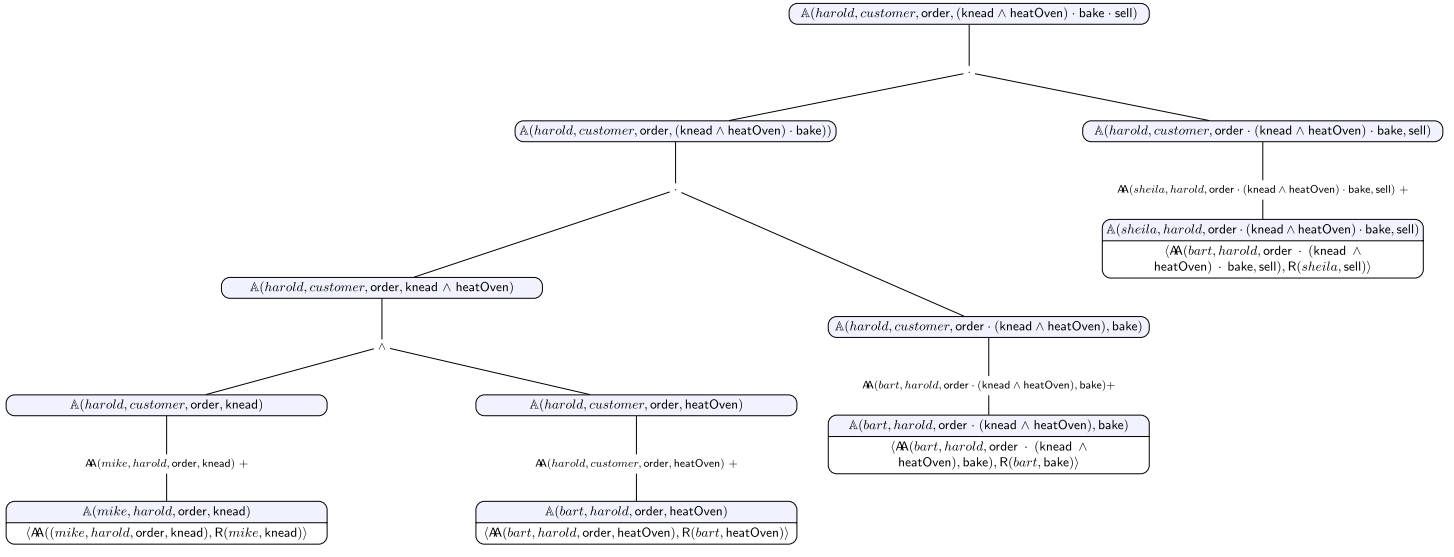

Fig. 2 The structural dimension underlying accountability $\mathbb{A}(harold, customer, \mathsf{order}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell})$

In the general case, u is $u_1 \wedge u_2$, then $\mathbb{A}(x, y, r, u_1 \wedge u_2) = \mathbb{A}(x, y, r, u_1) \wedge \mathbb{A}(x, y, r, u_2)$.

Since by hypothesis $u[u']$, then we have either $u_1[u']$ or $u_2[u']$.

Let us assume $u_1[u']$. By inductive hypothesis, for any workflow u' such that $u_1[u']$ there exists $\langle \mathbb{A}(x, y, r, u_1), \ldots, \mathbb{A}(x', y', r', u') \rangle$.

Now, if we substitute the first element of this chain with $\mathbb{A}(x, y, r, u)$, we get again feedback chain. In fact, by initial hypothesis $\mathbb{A}(x, y, r, u)$ is an accountability, and by construction $u[u_1[u']]$ holds.

The cases $\vee$ and $\cdot$ are similar.

Finally, in case $\mathbb{A}(x, y, r, u) = \mathbb{AA}(x, y, r, u) + \mathbb{A}(z, x, r, u)$, we have by inductive hypothesis, for any workflow u' such that $u[u']$ there exists $\langle \mathbb{A}(z, x, r, u), \ldots, \mathbb{A}(x', y', r', u') \rangle$.

Thus, the sequence $\langle \mathbb{A}(x, y, r, u), \mathbb{A}(z, x, r, u), \ldots, \mathbb{A}(x', y', r', u') \rangle$ is again a feedback chain, and this proves the proposition. □

Example 3 With reference to Example 2, thanks to $\mathbb{A}(harold, customer, \mathsf{order}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell})$ over $\mathbf{R}$, the workflow turns out to be an accountable workflow. For each subworkflow, due to the responsibilities in $\mathbf{R}$ and to the structure imposed by the accountability, it is possible to find a suitable feedback chain. For instance, for bake, the feedback chain is:

$\langle \mathbb{A}(harold, customer, \mathsf{order}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell}),$

$\mathbb{A}(harold, customer, \mathsf{order}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake}),$

$\mathbb{A}(harold, customer, \mathsf{order} \cdot (\mathsf{knead} \wedge \mathsf{heatOven}), \mathsf{bake}), \mathbb{A}(bart, harold, \mathsf{order} \cdot (\mathsf{knead} \wedge \mathsf{heatOven}), \mathsf{bake}) \rangle$, as depicted in Fig. 2.

Accountability is preserved at runtime with respect to the events that occur during the execution.

Proposition 2 (Accountability persistency) *Let u be a workflow and let $\mathbf{R}_u$ be a responsibility distribution. Given $\mathbb{A}(x, y, r, u)$ over $\mathbf{R}_u$ and an event e such that $r/e \neq 0$, we have that $\mathbb{A}(x, y, r/e, u)$ holds over $\mathbf{R}_u$.*

Proof The proof is by induction on the structure of $\mathbb{A}(x, y, r, u)$.

In the base case, $\mathbb{A}(x, y, r, u)$ is an A-leaf $\langle \{\mathbb{AA}(x, y, r, u)\}, \{\mathsf{R}(x, u)\} \rangle$. By definition of progression on $\mathbb{AA}(\cdot)$, it follows that $\mathbb{A}(x, y, r, u)/e = \langle \{\mathbb{AA}(x, y, r/e, u)\}, \{\mathsf{R}(x, u)\} \rangle = \mathbb{A}(x, y, r/e, u)$: the awareness of the agent x does not change after the progression of the context r under the occurrence of event e.

In the general case, we distinguish two situations. First, let $\mathbb{A}(x, y, r, u)$ be $\mathbb{A}(x, y, r, u')$ *op* $\mathbb{A}(x, y, r, u'')$ (where $op \in \{\vee, \wedge, \cdot\}$), and by inductive hypothesis let $\mathbb{A}(x, y, r/e, u')$ and $\mathbb{A}(x, y, r/e, u'')$ hold. It follows immediately that also $\mathbb{A}(x, y, r/e, u)$ holds. Second, let $\mathbb{A}(x, y, r, u)$ be $\mathbb{AA}(x, y, r, u) + \mathbb{A}(z, x, r, u)$, also in this case the thesis follows directly by the inductive hypothesis $\mathbb{AA}(x, y, r/e, u)$ and $\mathbb{A}(z, x, r/e, u)$. □

The proposition assures that, when r is a complex condition, the progression of r against an event e (such that $r/e \neq 0$) does not invalidate $\mathbb{A}(x, y, r, u)$. That is, any residual expression r/e is still an enabling condition for the accountability. Eventually r/e progresses to $\top$, allowing y to request x an account about u.

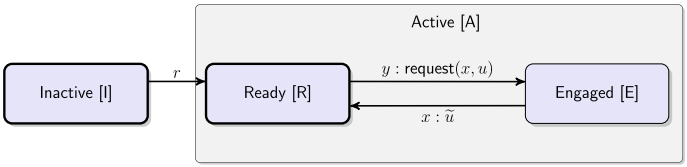

Fig. 3 Lifecycle of accountability $\mathbb{A}(x, y, r, u)$

3.2 Accountability lifecycle

In order to understand the implications of accountability, depending on contextual conditions, it is useful to associate to each $\mathbb{A}(x, y, r, u)$ a state, that may change depending on the occurrence of events. Figure 3 shows the accountability lifecycle that we have devised. This lifecycle assumes the existence of an agreement between the two agents x and y, and assumes also that such an agreement is persistent throughout the execution.

Figure 3 highlights that accountability $\mathbb{A}(x, y, r, u)$ has two main states, Inactive and Active, and that the latter is decomposed into the two substates Ready and Engaged. An accountability remains Inactive until condition r occurs. From a normative perspective, this implies that the a-taker y does not hold the right to request an account. Once condition r occurs, the accountability gets Active, and in particular Ready. Now the a-taker has the right to place a request, and when this happens, the accountability evolves into the Engaged state. Such a state captures the obligation over the a-giver x to provide an account $\widetilde{u}$ about the workflow u. Providing the account leads the accountability back to Ready: an accountability relationship is not resolved just because an account is provided, rather the a-taker keeps the right to ask for an account as far as the accountability remains Active. Ready and Inactive are an acceptance states (bold frames). Intuitively, we expect that whenever the two agents terminate properly (or leave the organization), their accountability are in one of these two states. The rationale is that in these two states there are no pending obligations that need to be fulfilled.

Finally, it is worth noting that condition r may, or may not, be under the control of the agents, depending on the domain at hand. So, it may happen that an a-taker acquires the right of asking for an account due to the occurrence of events in the environment. Under this respect, in Sect. 5.2 we will discuss how the accountability lifecycle can be related to the lifecycle of goals by means of some general practical rules that can help programming agents while leveraging accountability.

4 Robustness upon accountability

Robustness is a property that a system has, or has not. It concerns a perturbation and requires the existence of appropriate *handlers*, i.e., recovery strategies foreseen at an organizational level, which the agent, receiving the perturbation account, should activate for bringing the system to an acceptable state. Rephrasing [1], a process u is robust to some perturbation, when u includes some kind of recovery that allows u to terminate smoothly despite the occurrence of the perturbation (i.e., to terminate leaving the system in a "consistent" state). We will see that, in order to allow recovery, agents will produce and return accounts that witness what happened to disrupt the execution.

Definition 10 *(Account)* Given a workflow u:

- An *account* for u is a sequence of events $\widetilde{u} = (e_1, e_2, \ldots, e_n)$, such that for each e_i, $u/e_i \neq u$ (the event e_i is relevant for u).
- An account $\widetilde{u_!} = (e_1, e_2, \ldots, e_n)$ witnesses a perturbation for u when $u/(e_1, e_2, \ldots, e_n) = 0$.

We denote as $\overline{u_!}$ a recovery event based on the account $\widetilde{u_!}$. Such an event abstracts any workflow whose execution overcomes the effects of the perturbation witnessed by $\widetilde{u_!}$.

With reference to the model in Fig. 1, $\overline{u_!}$ amounts to the outcome of the `Treatment Task`, associated with the `Account Template`. Note that an account is not the perturbation it witnesses (i.e., an account does not identify the perturbation); if needed, perturbations could be identified by diagnostic reasoning like [43].

Proposition 3 (below) states that a workflow is robust to a perturbation, when it includes a recovery strategy, that can deal with the perturbation account so as to bring the workflow to an acceptable termination state. Formally, a *recovery strategy* for a workflow u for a perturbation with account $\widetilde{u_!} = (e_1, \ldots, e_n)$ is the workflow $h(\widetilde{u_!}) = e_1 \cdot \ldots \cdot e_n \cdot \overline{u_!}$. If the workflow u is perturbed with the account $\widetilde{u_!}$, the workflow $u \vee h(\widetilde{u_!})$ is not. More importantly, $u \vee h(\widetilde{u_!})/(\widetilde{u_!}, \overline{u_!}) = \top$; that is, when a perturbation is followed by an account describing its context and a suitable recovery event, the system terminates in a consistent state. More generally, we can prove the following proposition.

Proposition 3 (Robustness to a perturbation) *Let u be a workflow such that $u[v]$, where v is perturbed with account $\widetilde{v_!}$. Let w and z, be two sequences of events, such that $u/wtz = \top$ and $v/t = \top$, for some sequence t. We have that $(u \vee h(\widetilde{v_!}))/wt'z = \top$, where $t' = (\widetilde{v_!}, \overline{v_!})$.*

Proof The proof is by induction on the structure of u.

If $u \equiv v$, then w and z are empty, and we just need to prove that $(v \vee h(\widetilde{v_!}))/t' = \top$.

This, however, follows directly from the definition of recovery strategy since $h(\widetilde{v_!}) = e_1 \cdot \ldots \cdot e_n \cdot \overline{v_!} = \widetilde{v_!} \cdot \overline{v_!}$, and $t' = (\widetilde{v_!}, \overline{v_!})$ by hypothesis. Thus, $(v \vee h(\widetilde{v_!}))/t' = v/t' \vee h(\widetilde{v_!})/t'$; the first disjunct progresses to 0 whereas the second to $\top$.

In the general case, $v \not\equiv u$, and $u[v]$. Suppose that $u = a \wedge u'$ and $u'[v]$. By inductive hypothesis $(u' \vee h(\widetilde{v_!}))/w't'z = \top$, where $w = aw'$. By progression, we have $(a \wedge u' \vee h(\widetilde{v_!}))/aw't'z = (a \wedge u')/aw't'z \vee h(\widetilde{v_!})/aw't'z = (a/aw't'z \wedge u'/aw't'z) \vee h(\widetilde{v_!})/aw't'z = (\top \wedge u'/w't'z) \vee h(\widetilde{v_!})/w't'z = u'/w't'z \vee h(\widetilde{v_!})/w't'z = (u' \vee h(\widetilde{e_!}))/w't'z$. By inductive hypothesis $(u' \vee h(\widetilde{v_!}))/w't'z = \top$.

The cases $\vee$ and $\cdot$ are similar. The proof can be generalized for any workflow preceding and following u' in u. □

It is worth noting that, when the perturbation does not occur, we still have that $(u \vee h(\widetilde{v_!}))/wtz = \top$ because $u/wtz = \top$. In the above proposition, we have made the simplifying assumption that $h(\widetilde{v_!})$ is not perturbed by other perturbations. This assumption can be relaxed by introducing further recovery strategies, one for each perturbation affecting $h(\widetilde{v_!})$. This could be repeated indefinitely specifying recovery strategies for perturbations affecting recovery strategies. In practice, as usual happens in the design of complex systems, the designer has to put a limit to the depth of this chain of perturbation handlers.

Proposition 2 assures that accountability is preserved against progression. Intuitively, when a workflow is robust to a perturbation with account $\widetilde{v_!}$, not only there exists a specific

recovery strategy $h(\widetilde{v_!}) = \widetilde{v_!} \cdot \overline{v_!}$, but, thanks to accountability, it is also guaranteed that the account of the perturbation will actually be available.

By providing the account $\widetilde{v_!}$ we get two results: first, it is possible to notify the agent that will take care of the treatment that something wrong has occurred, and appropriate action is needed; second, it is possible to provide the same agent with information that helps understand how to handle the situation. This is important when agents, as often happens, do not have a complete view of events.

Proposition 4 (Account Availability) *Let $u = u' \vee h(\widetilde{v_!})$ be a workflow such that $u'[v]$, and let $\widetilde{v_!}$ be the account for a perturbation on v. Let $\mathbf{R}_u$ be a distribution of responsibility, such that $\mathsf{R}(y, u)$ belongs to $\mathbf{R}_u$. If u' is an accountable workflow through $\mathbb{A}(x, y, r, u')$ over $\mathbf{R}_u$, then the account is available to the agent y in charge of the recovery strategy for the perturbation.*

Proof The proof follows directly from Propositions 3 and 1. From Proposition 3, we have that a workflow u' such that $u'[v]$ is robust to a perturbation with account $\widetilde{v_!}$ if a dedicated recovery strategy $h(\widetilde{v_!}) = \widetilde{v_!} \cdot \overline{v_!}$ is activated instead of u' when $\widetilde{v_!}$ occurs. On the other hand, the account $\widetilde{v_!}$ must be generated somewhere in the system. This is granted by Proposition 1: since u' is an accountable workflow, there is always the chance to generate an account for any of its sub-workflow, including v, even when it is perturbed. There exists, in fact, a feedback chain $\langle \mathbb{A}(x, y, r, u'), \ldots, \mathbb{A}(x', y', r', v) \rangle$, where the account $\widetilde{v_!}$ is first generated by x' and then propagated to y, responsible for the whole workflow $u = u' \vee h(\widetilde{v_!})$.

□

Example 4 Let us consider a perturbation ovenBroken, concerning bake, and a corresponding recovery event scheduleOvenRepair. We can extend the responsibility distribution $\mathbf{R}$ so as to make the production workflow robust by adding the responsibility $\mathsf{R}(harold, \mathsf{scheduleOvenRepair})$ and turning the workflow into $(\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell}) \vee \mathsf{scheduleOvenRepair})$ – *harold* will also be in charge of the treatment.

Should a perturbation ovenBroken occur, by way of the new responsibilities, an account would be provided by *bart* to *harold*; this, in turn, would exploit the account for activating a recovery strategy.

5 Extending JaCaMo with accountability

We now show how accountability can be used in practice by describing an extension[1] of the JaCaMo [16] agent platform that encompasses it, and by showing the interplay between accountability and goals, reporting six typical schemes that we have identified, that will be used, in the following section, to program the agents.

Briefly, JaCaMo is a conceptual model and programming platform that integrates agents (programmed in Jason [18]), environments (programmed in CArtAgO [46]) and organizations (programmed in MOISE [39]). A MOISE organization has three dimensions. JaCaMo's structural dimension specifies roles, groups and links between roles in the

[1] Available at http://di.unito.it/moiseaccountability.

organization. JaCaMo's functional dimension is made of schemes, which elicit how the global organizational goal is decomposed into subgoals, and how subgoals are grouped in coherent sets, called missions – to be distributed to the agents. JaCaMo's normative dimension binds the two previous dimensions by specifying permissions and obligations that are associated with each role. At the beginning of the execution the agents, playing the different roles, are asked to commit to certain missions, as specified by the norms. Then, in order to coordinate the distributed execution, the organization will issue obligations to the committed agents to achieve mission goals.

A Jason agent is composed of a set of *beliefs*, i.e., predicates representing the agent's current state and knowledge about the environment, a set of *goals*, which correspond to tasks the agent can perform, and a set of *plans*, i.e., courses of action that are triggered by events and are pursued by executing plans. It is possible to specify both achievement ('!') and test ('?') goals; moreover, goals can be either organizational or local. In the first case, the goal is part of a schema (functional decomposition) of the organization, and is marked as achieved by an explicit `goalAchieved(...)` operation within an agent plan. A plan has the form:

$$triggering\_event : \langle context \rangle \leftarrow \langle body \rangle$$

where *triggering_event* denotes the event that the plan handles (a belief/goal addition or deletion), *context* specifies the circumstances in which the plan can be used, the *body* expresses a course of actions.

We map the concepts of `Responsibility` and `Task`, from the conceptual model in Fig. 1, to the concepts *Mission* and *Goal* of JaCaMo. We interpret the agent's commitment to a mission as an assumption of responsibility. Indeed, in JaCaMo agents commit to missions before pursuing the organizational goal and, from that moment on, the organization can issue obligations towards them to make them achieve the mission goals.

5.1 Encompassing accountability

In order to encompass accountability in JaCaMo, we have extended the XML specification of a MOISE organization, so as to include accountability agreements thanks to a new set of tags. The following piece of code shows how such an extension looks like.

```
<role-definitions>
    <role id="X"> <role id="Y"> ...
</role-definitions>

<accountability-agreement id="aa1">
    <concerns id="u" />
    <requesting-condition value="r" />
    <account-template>
        <account-argument id="..." arity="..." />
        <goal id="REQ_G" atype="requesting" /> <!-- Requesting Goal -->
        <goal id="ACC_G" atype="accounting" /> <!-- Accounting Goal -->
        <goal id="TRE_G" atype="treatment" when="..." /> <!-- Treatment Goal -->
    </account-template>
</accountability-agreement>

<mission id="m1" min="..." max="...">
    <goal id="ACC_G" /> ...
</mission>
<mission id="m2" min="..." max="...">
    <goal id="REQ_G" /> <goal id="TRE_G" /> ...
</mission>
...

<norm id="n1" type="obligation" role="X" mission="m1" />
<norm id="n2" type="obligation" role="Y" mission="m2" />
```

Note how the XML tags reflect, as far as possible, the structures and relations that we have depicted in the conceptual model of Fig. 1. In particular, we can see how the goals involved in an accountability agreement (requesting, accounting and treatment goal) appear inside JaCaMo missions, and norms are added to tie roles to such missions: the agents playing such roles are obliged to commit to the associated missions. For the sake of readability, we will use the following human-friendly representation of accountability agreement instead of using XML:

```
Accountability Agreement id: aa1
  concerns: u
  requesting condition: r

  Account Template:
    must account with: arguments
    (requesting goal: REQ_G
       [when: cond])*
    accounting goal: ACC_G
       [when: cond]
    (treatment goal: TRE_G
       [when: cond])*
```

It is worth noting how the above listing creates the connection among the model, that was presented in Fig. 1, the formalization of agreements and accountability, and their implementation in JaCaMo. The listing, indeed, represents an accountability agreement $\mathbb{AA}(x, y, r, u)$. Fields **concerns** and **requesting condition** match directly the namesake relationships of the model. The former refers to the goal u, whereas the latter refers to condition r. Note that in Sect. 3 u is a formula that specifies how a complex task is decomposed into

atomic tasks, this is mapped in JaCaMo to the functional decomposition of complex goal u. Requesting condition r, instead, represents what events must occur to activate accountability. In JaCaMo, r is mapped to organizational events such as the achievement of goals, the issue of obligations, the failure of goals, and so on. Section `Account Template` specifies three aspects referring to accounts: how an account can be requested, how it can be provided (and with which format), and, optionally, how it can be treated. For all these three aspects we exploit JaCaMo goals. **requesting goal** denotes an organizational goal whose satisfaction assumes the social meaning of a request for an account about u, thus only a legitimate a-taker can pursue such a goal. Note that each `Account Template` can include zero to many requesting goals. If no requesting goal is specified, an implicit account request will be placed via the normative system of the organization as soon as requesting condition becomes true. The field **accounting goal** denotes an organizational goal that becomes enabled when a request has been done, and hence a corresponding obligation is issued upon the appropriate a-giver. The a-giver will pursue such a goal by issuing an account, shaped according to the field **must account with**: a list of arguments that form the account. An account is therefore a structured piece of information that an agent make available to others. In principle, an `Account Template` could maintain several accounting goals as different ways for producing an account. In our current implementation, we can specify one **accounting goal**, and leave the extension to a future development.

Finally, **treatment goal** is an optional field that can be specified when the treatment to some account is prescribed by design constraints. This means that the way for handling an account is not left to the local decision of an agent, but it is an organizational goal; we will better discuss this point in some examples below. More treatment goals can be specified for a single account to capture alternative ways to cope with the account itself.

One can notice that neither the a-taker x nor the a-giver y involved in the accountability agreement are directly mentioned within the XML tag. This happens because we rely on the structures already defined in JaCaMo, and in particular on JaCaMo roles and missions. In fact, the goals mentioned within the XML tag must be included, as any other organizational goal in JaCaMo, within a mission, to which agents playing roles have to commit. The commitment to a mission creates, thus, the binding between a goal and the agent responsible for that goal. The optional field **when** associated with each goal in the account template represents an applicability condition that the organization designer may be willing to specify.

Accountability agreements capture the normative dimension of accountability. More precisely, each accountability agreement is translated into a set of norms that will be integrated within the normative program followed by the organization. These norms will yield permissions and obligations about accounts during the system execution. We refer to [13] for a detailed description on how an accountability agreement is translated into JaCaMo norms. In order to guarantee the availability of sound accounts, an organization designer must take care of defining a proper set of agreements, so as to capture the structural dimension, as well. An automated tool to verify the structural consistency of a given set of accountability agreements is discussed in [9].

5.2 Interplay between goals and accountability

From an agent programming perspective, it is useful to make the relationship between accountability and goals explicit, so as to guide the programmer in the development of "accountable" agents, and in particular when agents have to decide how to cope with

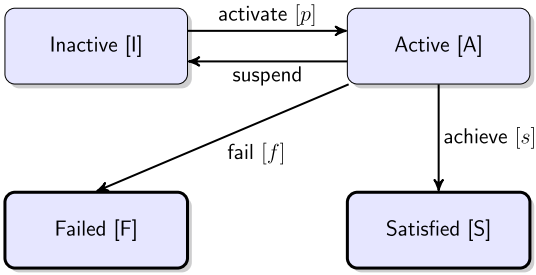

Fig. 4 Goal $\mathsf{G}(x,p,s,f)$ lifecycle [34]

perturbations and opportunities. Account request and account production amount to goals; thus, accountability has an impact on the goals that agents pursue. To this end, we first briefly formalize the notion of goal lifecycle, and then relate such a lifecycle with that of accountability by means some practical rules directly implemented as JaCaMo plans.

Generally speaking, a goal represents a condition that an agent wants to achieve. Many works in literature (e.g., [34, 50–53]) have proposed several formalizations to serve as a basis for mechanisms of goal reasoning. In this paper, we take as a reference the formalization proposed in [50, 53], where a goal is modeled as a structure $\mathsf{G}(x,p,v,q,s,f)$ denoting: x the agent responsible for the goal; p a precondition that must be true before G can become Active and hence pursued by x; v an invariant condition that is true once G is Active until its achievement; q a post-condition that holds when G is successfully achieved. Finally, s and f specify the success and failure conditions, respectively, of G. Such a rich representation of goals allows agents to reason about their objectives. In JaCaMo, however, there is not a substantial distinction between q and s, and the concept of invariant condition v is not native. Thus, to simplify the discussion, in the following we consider goals shaped as $\mathsf{G}(x,p,s,f)$; however, our approach allows one to take full advantage of v and q, as well, when these conditions are directly available in the framework at hand. Several lifecycles have been proposed to capture relevant state changes in goals. In this paper, we take as a reference the lifecycle introduced in [34] showed in Fig. 4. A goal is Inactive if its preconditions do not hold. When inactive, a goal cannot be pursued by agent x. A goal is Active when its preconditions hold and neither satisfaction nor failure holds. This means that agent x is pursuing such a goal. A goal is Satisfied when condition s holds. Whereas, a goal is Failed, if failure occurs. An active goal may also be suspended, if the precondition stops to hold. The Satisfied and Failed states are terminal states.

We are now in the position for introducing some practical rules that relate accountability and goals. These rules highlight how accountability actually affects agent behaviors: a change in the state of one accountability induces a change in the one or more goals of the corresponding a-taker and a-giver. These rules are independent of any specific agent platform at hand, but play a central role in programming the agents. To emphasize this, we map each rule into a Jason template (involving two or more plans), that represents a possible way to implement the rule. In Sect. 6 we will show how these templates can be used as building blocks for solving recurrent interaction problems of distributed systems.

The *Be-Accountable* rule

$$\mathbb{A}^E(x,y,r,u) \Rightarrow \mathsf{G}^A(x,p,\widetilde{u},f)$$

states that when an accountability $\mathbb{A}(x,y,r,u)$ gets *engaged*, the a-giver x has to activate an internal goal that produces the asked account $\widetilde{u}$ about workflow u. The rationale is that,

to be accountable, an agent has to pursue the goal of producing an account when a legitimate request is placed. Since $\mathbb{A}(x, y, r, u)$ is part of the organization state, and modeled by means of dedicated norms, the organization acts as a mediator between the a-taker and the a-giver: the a-taker's request is placed by accomplishing an organizational `Requesting Task` (see Fig. 1). The normative system, thus, progresses the accountability into the *engaged* state, and generates an obligation to produce an account targeted on the a-giver. Such an obligation is justified by virtue of the agreement, between the a-taker and a-giver and implicitly maintained in $\mathbb{A}(x, y, r, u)$, for which the a-giver assumes the responsibility to produce an account when asked for. To fulfill this obligation, the a-giver needs to accomplish an organizational `Accounting Task`, that is mapped into an internal goal $\mathsf{G}(x, p, \widetilde{u}, f)$. Of course, it is expected that the obligation to achieve the goal is issued only when precondition p holds. The following Jason template captures this behavior.

```
1 +obligation(X,<accounting-goal>) : my_name(X)
2   <- !<agent-accounting-goal>;
3      goalAchieved(<accounting-goal>).
4
5 +!<agent-accounting-goal>
6   <- <do something to produce the account>
7      giveAccount(<account>).
```

The first plan is triggered when the agent receives an obligation to provide an account by accomplishing the organizational goal *accounting-goal*. To do so, the agent activates an internal *agent-accounting-goal* whose plan ends up by producing an account by means of operation `giveAccount(<account>)`, which changes the state of the organization with the addition of a new fact, as an observable properties, in one of the organizational artifacts. The achievement of the internal goal, thus, entitles the agent to mark the organizational goal as achieved, too.

The *Auditing* rule

$$\mathsf{G}^I(y, \widetilde{u}, s, f) \wedge \mathbb{A}^R(x, y, r, u) \Rightarrow \mathsf{G}^A(y, p, \textit{requestAccountOn-}u, f)$$

states that, given an inactive goal $\mathsf{G}(y, \widetilde{u}, s, f)$ and a ready accountability $\mathbb{A}(x, y, r, u)$ (i.e., y has the permission to ask x about u), then y can make $\mathsf{G}(y, \widetilde{u}, s, f)$ active by activating goal $\mathsf{G}(y, p, \textit{requestAccountOn-}u, f)$, provided that condition *cxt* holds. This goal amounts to a `Requesting Task`, and hence to transition $y : \mathsf{request}(x, u)$ in the accountability lifecycle (Fig. 3). The Jason template mapping this behavior is as follows:

```
1 +!<goal-to-s> : <account-ũ needed> & r
2   <- goalAchieved(<requestAccountOn-u>);
3      .wait(<account ready>);
4      !<goal-to-s>.
5
6 +!<goal-to-s> : <account ready>
7   <- <do something to achieve condition s>.
```

The first plan captures the situation in which agent y needs to achieve condition s, but the goal $\mathsf{G}(y, \widetilde{u}, s, f)$ cannot be activated due to the lack of information $\widetilde{u}$ (see context condition `<account-ũ needed>`). However, condition r holds and hence the normative system has granted y to achieve the organizational goal *requestAccountOn-u*. The body of this

plan, thus, consists of marking *requestAccountOn-u* as achieved (a request is placed), and, then, waiting for the corresponding answer. When the answer is ready, y achieve condition s by invoking the second plan.

The *Activate* rule

$$\mathsf{G}^I(y, \widetilde{u}, s, f') \wedge \mathbb{A}^I(x, y, r, u) \Rightarrow \mathsf{G}^A(y, p, r, f'')$$

is applicable only when agent y has control over condition r, activating accountability relationship $\mathbb{A}(x, y, r, u)$. The rationale is that, when y wants to activate a goal $\mathsf{G}(y, \widetilde{u}, s, f')$, as before, but the accountability is Inactive, then the agent can activate goal $\mathsf{G}(y, p, r, f'')$ in order to make $\mathbb{A}(x, y, r, u)$ progress to the Active state. In the next section we points out how this rule applies when agent y needs further information (i.e., the account $\widetilde{u}$), to handle a specific situation, such as a perturbation or an opportunity. The following Jason template addresses this situation in a similar way as before. First, one specifies a plan for achieving condition s when an account is needed and the condition r does not hold. The body of this plan consists in the activation of another goal, *achieveRequestingCondition-r*, that aims at getting condition r. Reasonably the body of this second plan will contain some organizational goal since condition r must be true within the organizational state in order to activate the accountability. Once *achieveRequestingCondition-r* is accomplished, the agent can try to get condition s. To do so, the previous Jason template for *Auditing* can be applied.

```
1 +!<goal-to-s> : <account-ũ needed> & not r
2   <- !<achieveRequestingCondition-r>;
3      !<goal-to-s>.
4
5 +!<achieveRequestingCondition-r>
6   <- <do something to make r hold>.
```

The *Alert* rule

$$\mathsf{G}^F(x, t, u, f') \wedge \mathbb{A}^A(x, y, r, u) \Rightarrow \mathsf{G}^A(x, p, \widetilde{u}, f'')$$

captures the situation in which an agent is requested to provide an account about the failure of an organizational goal, which can be seen as perturbation affecting the normal, expected behavior. The rationale is that the goal failure, that calls for special treatments, is not necessarily observed directly by the agent who will handle that failure. To fill this gap, the account is implicitly asked, through the organization, so as to enable the failure treatment. This means that the normative system of the organization will issue an obligation to provide an account for a failure as soon as the very same failure is signaled by an agent. Of course, an accountability must exist between the agent x, detecting the failure (a-giver), and the agent y, handling the failure (a-taker). In addition, such an accountability must be Active. The a-giver, thus, activates a goal that will produce an account $\widetilde{u}$ about the failed goal. In Jason, the rule is implemented by the following template.

```
-!<organizational-goal> : r & f'
  <- goalFailed(<organizational-goal>).

+obligation(X,<accounting-goal>) : my_name(X)
  <- !<agent-accounting-goal>;
     goalAchieved(<accounting-goal>).

+!<agent-accounting-goal>
  <- <do something to produce the account>
     giveAccount(<account>).
```

The first plan represents the failure of an organizational goal: the trigger event is the deletion of the goal and the body consists in marking that goal as failed when the failure condition f' holds. Note that operation `goalFailed()` is part of our JaCaMo extension since in standard JaCaMo goals cannot marked as failed by the agents.

The second plan captures the obligation (i.e., the implicit request generated by the normative system) to provide an account about the failed goal. Specifically, the obligation is about the achievement of organizational goal *accounting-goal*. To accomplish such a goal, the agent pursues the local goal *agent-accounting-goal*, handled by the third plan. This last plan produces the account, and makes it available to the other agents in the organization by means of operation `giveAccount()`. Goal *accounting-goal* can therefore be marked as *achieved*, and the normative system will progress the state of the organization by issuing an obligation on agent y responsible for the failure treatment.

The *Treatment* rule

$$\widetilde{u} \Rightarrow \mathsf{G}^{A}(y, \widetilde{u}, s, f)$$

describes the complementary situation of the previous rule. Namely, an agent that has access to an account $\widetilde{u}$ is, then, asked to achieve an organizational goal that amounts to the treatment for such an account. In this case, the current state of $\mathbb{A}(x, y, r, u)$ is irrelevant, and hence $\mathbb{A}(\cdot)$ is not included in the head of the rule. The point is that $\mathbb{A}(\cdot)$ has already played its role for the production of $\widetilde{u}$, and now $\widetilde{u}$ needs to be addressed. The Jason template corresponding to this situation is as follows.

```
+obligation(Y,<treatment-goal>) : my_name(Y)
  <- !<agent-treatment-goal>;
     goalAchieved(<treatment-goal>).

+!<agent-treatment-goal> : account(<account>)
  <- <do something to treat the account>.
```

The first plan allows the agent to intercept an obligation about the organizational goal *treatment-goal*. As in the previous case, this organizational goal is mapped into an agent goal, namely *agent-treatment-goal*, which is pursued by the second plan in the template. The completion of the local goal enables the agent to mark the organizational goal as achieved.

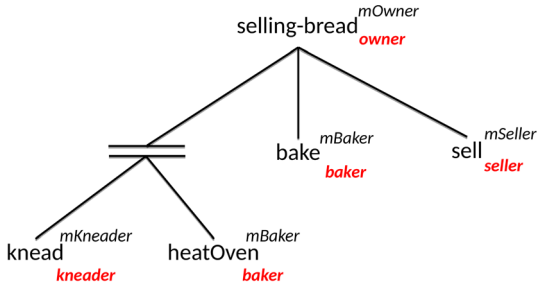

Fig. 5 Functional decomposition of the organizational goal in the Information Gathering scenario. Over each goal the name of the mission, to which the goal belongs, is reported. In red, we show the names of the roles whom the mission is assigned to

6 Agent programming

We now illustrate how the rules above can be used to practically program agents in JaCaMo. We will describe three scenarios, that capture a number of typical situations. Each scenario will be introduced by explaining the general problem and highlighting which rules can be used to solve it. Then, we get into the details of agent programming.

6.1 Information gathering

Problem: An agent needs a piece of information in order to take a decision. The information is not directly accessible to it; however, by way of an appropriate feedback chain specified by the organization designer, the agent can rely on fellow agents in the organization in order to retrieve it.

Context: Accountability $\mathbb{A}(x, y, r, u)$ is defined in the organization between y, the agent needing the information about u, here acting as a-taker, and x, which can account for u, here acting as a-giver. Either condition r already holds, so the accountability is in the state Ready of its lifecycle (see Sect. 3.2), or the accountability is Inactive and agent y has the possibility to make r hold.

Rules: *Activate* and *Auditing* for the a-taker, and *Be-Accountable* for the a-giver.

Exemplification: Let us consider Example 1, where *harold* is the bakery owner. In JaCaMo the workflow $(\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake} \cdot \mathsf{sell}$ amounts to the organizational goal. It is the root of the functional decomposition in Fig. 5, where it is identified by the label selling-bread. The organizational goal is decomposed into four subgoals, which are under the responsibility of three different agents (*sheila*, *bart*, *mike*). This happens because such agents play the roles *seller*, *baker* and *kneader* respectively. In JaCaMo, missions contain the goals from the functional decomposition, and are assigned to roles through norms. At runtime, agents playing such roles are required to commit to the corresponding missions, taking responsibilities for mission goals. In this way, responsibilities are distributed in JaCaMo. In particular, as shown in Example 2, we have that the workflow is grounded over the following responsibility distribution $\mathbf{R} = \{\mathsf{R}(\mathit{mike}, \mathsf{knead}), \mathsf{R}(\mathit{bart}, \mathsf{heatOven}), \mathsf{R}(\mathit{bart}, \mathsf{bake}), \mathsf{R}(\mathit{sheila}, \mathsf{sell})\}$.

The following listing illustrates how *sheila*'s, *bart*'s and *mike*'s responsibilities are encoded, through roles, missions, and norms in the organizational specification.

```
<role-definitions>
    <role id="kneader"> <role id="baker"> <role id="seller"> ...
</role-definitions>

<mission id="mKneader" min="1" max="1">
    <goal id="knead" /> <goal id="notifyFlourTypeToBaker" />
</mission>
<mission id="mBaker" min="1" max="1">
    <goal id="heatOven" /> <goal id="bake" />
    <goal id="requestFlourTypeToKneader" /> <goal id="notifyFlourTypeToSeller" />
</mission>
<mission id="mSeller" min="1" max="1">
    <goal id="sell" /> <goal id="getAuthorization" />
    <goal id="requestFlourTypeToBaker" />
</mission>

<norm id="n1" type="obligation" role="kneader" mission="mKneader" />
<norm id="n2" type="obligation" role="baker" mission="mBaker" />
<norm id="n3" type="obligation" role="seller" mission="mSeller" />
```

Mission *mBaker*, for instance, contains goals heatOven and bake, and it is assigned to role *baker* through norm n2. At runtime, agent *bart* will play the *baker* role and, by committing to such a mission, he will take on the responsibility for the two goals. In a similar way, responsibilities are taken on by *sheila*, and *mike*.

Let us show how, by exploiting a feedback chain specified at design time via accountability, *sheila* can get pieces of information that are outside her scope, and use them in her local decision-making process. Let us assume that agent *sheila*, who sells the bread, is also in charge of setting the price depending on the type of flour that is used. However, *sheila* does not know what flour type was used, since she is not directly involved in the process of kneading and baking the bread. Two accountabilities, specified in the organization, can help *sheila*; $a_1 : \mathbb{A}(mike, bart, \top, \mathsf{knead})$, and $a_2 : \mathbb{A}(bart, sheila, \mathsf{getAuthorization}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake})$. Since the type of flour can influence the baking time, accountability a_1 allows *bart* to ask *mike* for an account about knead; the requesting condition $\top$ means that *bart* can always legitimately ask *mike* about knead. On the other hand, accountability a_2 allows *sheila* to ask for an account about the workflow producing the bread, which will include also information on the used type of flour. However, *sheila* can request an account provided that she gets the authorization by the owner: getAuthorization is an event that is generated as a consequence of the goals pursued by *sheila* herself. Following the same reasoning explained in Example 3, it is possible to show that the two accountabilities form a feedback chain, given the responsibility distribution $\mathbf{R}$, and hence the subworkflow $(\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake}$ is an accountable workflow (see Proposition 1). In fact, by means of a_2, *sheila* can have information on the bread production process from *bart*, which can in turn leverage a_1 to gather information from *mike* about the kneading part.

In JaCaMo, the two accountabilities a_1 and a_2 are specified by introducing the two following agreements:

```
Accountability Agreement id: a1
  concerns: knead
  requesting condition: true

  Account Template:
    must account with: flourType
    requesting goal: requestFlourTypeToKneader
    accounting goal: notifyFlourTypeToBaker
```

```
Accountability Agreement id: a2
  concerns: (knead ∧ heatOven) · bake
  requesting condition: satisfied(getAuthorization)

  Account Template:
    must account with: flourType
    requesting goal: requestFlourTypeToBaker
    accounting goal: notifyFlourTypeToSeller
```

The *a-taker* and *a-giver* agents are designated by including the requesting goals requestFlourTypeToKneader and requestFlourTypeToBaker in the missions assigned to the *baker* and *seller* roles, and the accounting goals notifyFlourTypeToBaker and notifyFlourTypeToSeller in the missions assigned to the *kneader* and *baker* roles, respectively (see the listing above). In addition, since the responsibility distribution $\mathbf{R}$ assigns each atomic task of the workflow (knead ∧ heatOven) · bake · sell to one of the three agents, both agreements are grounded on responsibility. Agreements and responsibilities together yield, thus, accountabilities as in Definition 8. For instance, the A-structure $\langle\{\text{AA}(mike, bart, \text{order}, \text{knead})\}, \{\text{R}(mike, \text{knead})\}\rangle$, representing accountability a_1, is encoded in our extended JaCaMo by agreement a1 above and the distribution of responsibilities encoded by missions. Accountability a_2 is also encoded within the system in a similar way.

Coming to agent programming, the *Activate* and *Auditing* rules easily guide the writing of *sheila*'s program, as follows.

```
1  +!sell
2     : not account(flourType(FT)) & not satisfied(getAuthorization)
3    <- <get the authorization>
4       goalAchieved(getAuthorization);
5       !sell.
6
7  +!sell
8     : not account(flourType(T))  & satisfied(getAuthorization)
9    <- goalAchieved(requestFlourTypeToBaker);
10      .wait({+account(flourType(_))});
11      !sell.
12
13 +!sell : account(flourType(organic))
14   <- <sell at a higher price>.
```

The first plan allows *sheila*to activate the accountability by making its requesting condition hold (*Activate* pattern). To this end, she pursues and sets the organizational goal getAuthorization as achieved. The other plans, instead, follow the *Auditing* pattern. The second one allows *sheila* to request for an account, provided none is already available; she

will do so while pursuing her own goal `sell`. The request is concretely performed by marking the requesting goal requestFlourType as achieved. The execution of the plan is then suspended until the account is made available. Once the needed information is provided, *sheila* attempts to pursue the sell goal again, likely selecting a different plan for execution. For instance, the last plan will be executed if the received account denotes the use of an organic flour. Other plans targeting different eventualities may be available, as well.

Conversely, the *Be accountable* rule can be applied to *bart*'s code, below (in combination with the *Auditing* one).

```
+!obligation(Ag,_,done(_,notifyFlourTypeToSeller,Ag),_) : my_name(Ag)
  <- !notifyFlourTypeToSeller
     goalAchieved(notifyFlourTypeToSeller).

+!notifyFlourTypeToSeller
   : account(flourType(FT))
  <- giveAccount([flourType(FT)]).

+!notifyFlourTypeToSeller
   : not account(flourType(FT))
  <- goalAchieved(requestFlourTypeToKneader);
     .wait({+account(flourType(_))});
     !notifyFlourTypeToSeller.
```

The request performed by *sheila* makes $a_2 : \mathbb{A}(bart, sheila, \mathsf{getAuthorization}, (\mathsf{knead} \wedge \mathsf{heatOven}) \cdot \mathsf{bake})$ become *engaged*, resulting in an obligation for *bart* to pursue the corresponding accounting goal. The first plan realizes this behavior by mapping such an obligation to an internal goal of the agent, which is, then, satisfied by means of the second and third plans. If the flour type is already available to the agent (second plan), the account is produced by means of the `giveAccount(...)` primitive, that is provided by our extended JaCaMo infrastructure. It is worth noting that the last two plans follow the *Auditing* rule, as well. If the flour type is not available to the agent, yet, the last plan allows *bart* to request an account to *mike*, in a similar way to what done by *sheila*.

Finally, agent *mike* can be programmed by simply following the *Be accountable* rule as follows.

```
+!obligation(Ag,_,done(_,notifyFlourTypeToBaker,Ag),_) : my_name(Ag)
  <- !notifyFlourTypeToBaker
     goalAchieved(notifyFlourTypeToBaker).

+!notifyFlourTypeToBaker
  <- <gather flour type T>;
     giveAccount([flourType(T)]).
```

6.2 Context-aware adaptation

Problem: An agent is interested in an event, which has an impact on the achievement of the agent's goals (e.g., it may represent an occasion the agent could profit, or some perturbation that may negatively impact on the efficiency by which the goal can be achieved). The occurrence of such an event may induce the agent to consider to change its behavior in order to adapt to the situation. The decision on if/how to react to the event depends on the

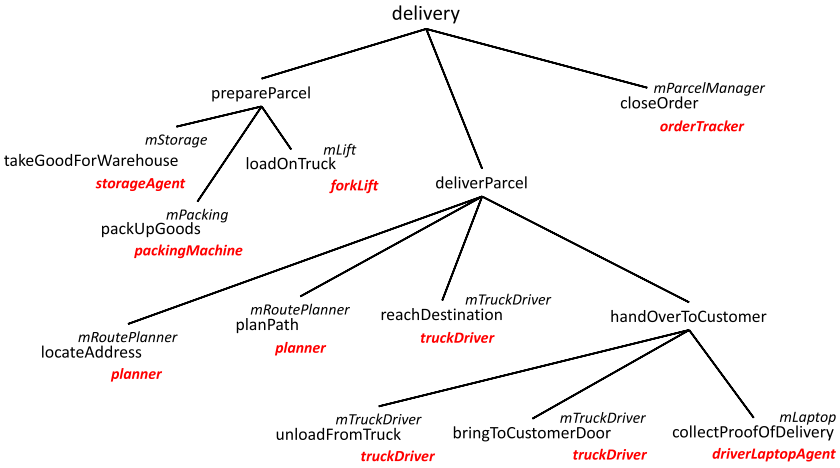

Fig. 6 Functional decomposition of the Context-Aware Adaptation scenario

context in which the event occurs, but this information is outside the scope of the agent; however, the agent can take advantage of the accountabilities of other agents, where it acts as a-taker.

Context: An accountability $\mathbb{A}(x, y, r, u)$ is defined in the organization between agent y, interested to know about u when event r occurs, and x which can account for u. Since event r has occurred the accountability is Ready.

Rules: *Auditing* for the a-taker and *Be-Accountable* for the a-giver.

Exemplification: Figure 6 shows the functional decomposition of an organization for the delivery of some goods. The process involves many agents playing multiple roles (in red) and carrying out several activities from goods packaging to shipping and delivery. We assume that the shipment is distributed and involves multiple trucks carrying the goods, e.g., to a construction site. In this case, a truck driver who realizes the delay of some fellow truck may likely be interested in avoiding the issue, if any, that the fellow truck has encountered.

Let us consider a case in which two agents (*alice* and *bob*) play the same role *truckDriver*. The two are responsible for reaching the destination with their respective trucks. We suppose that accountability $\mathbb{A}$(*alice*, *bob*, delay(reachDestination), reachDestination) holds; that is, should a delay[2] occur in the achievement of goal reachDestination, *bob* would have the right to ask *alice* an account, if interested. A complementary relationship, involving *bob* as a-giver and *alice* as a-taker holds, as well.

We can capture both accountabilities by means of the following accountability agreement, which also specifies that the account includes a reason for the delay, and a list of roads, that are concerned.

[2] delay(G) is a keyword encoding an organizational event of unfulfillment of an obligation concerning goal G, issued towards some responsible agent. We use it as a shortcut for the formula scheme_id(S) & unfulfilled(obligation(_,_,done(S,G,_),_)).

```
Accountability Agreement id: aaDelivery
  concerns: reachDestination
  requesting condition: delay(reachDestination)

Account Template:
  must account with: reason, roads
  requesting goal: requestDelayReason
  accounting goal: reportDelayReason
```

Since *alice* and *bob* play the same role in the organization, they will be able to act both as a-taker or a-giver, depending on the circumstance. To this end, both the requesting and the accounting goals are included in the mission assigned to the *truckDriver* role, that is played by the two agents, as follows.

```
1  <role-definitions> <role id="truckDriver"> ... </role-definitions>
```

```
2
3  <mission id="mTruckDriver" min="2" max="2">
4      <goal id="reachDestination" />
5      <goal id="requestDelayReason" />
6      <goal id="reportDelayReason" />
7  </mission>
8
9  <norm id="n1" type="obligation" role="truckDriver" mission="mTruckDriver" />
```

Here, the mission cardinality states that exactly two agents should commit to the mission because the two agents take part to the same delivery. Goal reachDestination will have the same cardinality, meaning that it will be considered as satisfied only when both agents will have achieved it.

The *Auditing* rule can be applied to realize the account taking behavior needed for adaptation. The following excerpt of code allows each of the two agents to ask for an account to the other and react accordingly.

```
+oblUnfulfilled(obligation(Ag,_,done(_,reachDestination,Ag),_))
    : not my_name(Ag) & not routeReplanned
  <- !investigateDelay.

+!investigateDelay
    : not account(reason(R))
  <- goalAchieved(requestDelayReason);
     .wait({+account(_)});
     !investigateDelay.

+!investigateDelay
    : account(reason(roadworks))
  <- !updateMap;
     +investigated;
     !deactivate.

+!updateMap
    : account(roads(l))
  <- <add roads l to black list>.
```

The first three plans result from the *Auditing* rule. Suppose one of the agents, e.g. *alice*, detects a delay (i.e., a perturbation) in the achievement of the organizational goal reachDestination by its partner *bob* (i.e., the unfulfillment of *bob*'s obligation). Such a delay activates the accountability agreement between the two, which allows *alice* to investigate the reasons of *bob*'s delay by pursuing the organizational goal requestDelayReason.

The fourth plan realizes the agent's recovery strategy to the new situation it became aware of. It is triggered once the account is available. In the example program some roads are added to a black list and avoided. In general, the agent program will contain many plans (i.e., recovery strategies), for tackling different situations. Finally, the account giving behavior of the two agents can be programmed by following the *Be Accountable* rule, as before.

```
+!obligation(Ag,_,done(_,reportDelayReason,Ag),_) : my_name(Ag)
  <- !reportDelayReason;
     goalAchieved(reportDelayReason).

+!reportDelayReason
  <- giveAccount([reason(roadworks), roads([mainStreet,fifthAvenue])]).
```

It is worth noting that the adaptation could be exploited not only in case of perturbations, but also in order to take advantage of opportunities that may arise during the execution. Suppose, for instance that one of the two drivers reaches the destination earlier than expected. The partner agent could be interested in investigating the reasons, e.g., to exploit the same low traffic roads. We can define an accountability agreement that is similar to the previous one.

```
Accountability Agreement id: aaDelivery
  concerns: reachDestination
  requesting condition: done(reachDestination)

Account Template:
  must account with: reason, roads
  requesting goal: requestQuicknessReason
  accounting goal: reportQuicknessReason
```

Here, the requesting condition denotes the successful achievement of goal reachDestination by some agent. In order to allow the agents take advantage of the opportunity, the plans, needed for adaptation, are to be slightly modified, as follows.

```
1  +oblFulfilled(obligation(Ag,_,done(_,reachDestination,Ag),_))
2     : not my_name(Ag) & not routeReplanned
3    <- !investigateQuickness.
4
5  +!investigateQuickness
6     : not account(reason(R))
7    <- goalAchieved(requestQuicknessReason);
8       .wait({+account(_)});
9       !investigateQuickness.
10
11 +!investigateQuickness
12    : account(reason(lowTraffic))
13   <- !takeShortcut;
14      +investigated;
15      !deactivate.
16
17 +!takeShortcut
18    : account(roads(I))
19   <- <leverage roads I to take a shortcut>.
```

Now, the first plan is triggered by an obligation fulfillment instead of an unfulfillment. By this, the a-taker can leverage the a-giver's experience to optimize its journey towards the destination. Adaptation, indeed, is realized by using the information in the account to take a low traffic shortcut.

6.3 Exception handling

Problem: In a MAO, the agent detecting an exceptional condition is not usually in the position for treating it. Accountability supports the realization of exception handling mechanisms: whenever an exceptional condition occurs, an account (i.e., exception) concerning that condition is reported *by default* to the agent(s) responsible for handling it. Terminologically, the a-giver can be identified as the *exception raiser*, the a-taker as the *exception handler*, and the account amounts to the raised exception.

Context: An accountability $\mathbb{A}(x, y, r, u)$ is defined in the organization between agent y, responsible for handling some exceptional condition r, that may occur during the execution of u, and agent x, which can provide the account $\widetilde{u}$. We assume that condition r holds (i.e., an exceptional condition has actually occurred), and that the account request has been implicitly sent, so the accountability is Active.

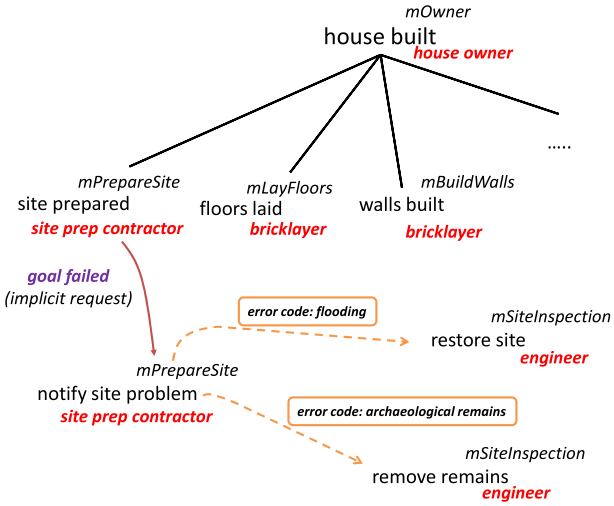

Fig. 7 Functional decomposition of the Exception Handling scenario

Rules: *Alert* for the a-giver (*exception raiser*), and *Treatment* for the a-taker (*exception handler*).

Exemplification: Let us consider the house building scenario described in [16]. Here, the organizational goal is to build a house on a plot, involving multiple companies that contribute on specific goals, as shown in Fig. 7. For instance, site preparation must be completed before any other step. Should a failure occur in the achievement of this goal, the whole house construction could not proceed. It is, thus, important to foresee a suitable strategy to deal with a possible failure of this goal. Indeed, depending on the reasons for the failure, different recovery actions could be applicable. We can effectively target this eventuality by specifying the following accountability agreement.

```
Accountability Agreement id: aaHouse
  concerns: sitePrepared
  requesting condition: failure(sitePrepared)

  Account Template:
    must account with: errorCode
    accounting goal: notifySiteProblem
    treatment goal: restoreSite
      when: errorCode(flooding)
    treatment goal: removeRemains
      when: errorCode(archaeologicalRemains)
```

The agreement specifies that an account (exception) must be provided (raised) every time a failure in the achievement of goal sitePrepared occurs, and it amounts to an error code. Two different treatments are included, to be applied depending on the provided error code.

By taking responsibility for the accounting goal notifySiteProblem, the *companyX* agent (playing the *site prep contractor role* and already in charge of site preparation) becomes exception raiser. Similarly, the agent playing the *engineer* role, by taking

responsibility for the treatment goals, can be identified as exception handler. The *Alert* rule guides the writing of *companyX*.

```
-!sitePrepared
  <- goalFailed(sitePrepared).

+obligation(Ag,_,done(_,notifySiteProblem,Ag),_) : my_name(Ag)
  <- !notifySiteProblem;
     goalAchieved(notifySiteProblem).

+!notifySiteProblem
  <- <retrieve failure reason R>;
     giveAccount([errorCode(R)]).
```

The first plan, triggered when the internal goal sitePrepared fails, notifies the failure to the organizational infrastructure through the `goalFailed(...)` primitive. The other plans are analogous to the ones foreseen by the *Be Accountable* rule. The only difference is that, the obligation to pursue the accounting goal is issued as soon as the failure is signaled within the organization, since an account request is assumed implicit.

Conversely, the following excerpt of code shows the *Treatment* rule applied to the *engineer* agent. Here two plans, targeting the two different treatment goals specified in the agreement, are defined.

```
+obligation(obligation(Ag,_,done(_,restoreSite,Ag),_))
   : my_name(Ag) & account(errorCode(flooding))
  <- <remove water and perform fix>;
     goalReleased(sitePrepared);
     goalAchieved(restoreSite).

+obligation(obligation(Ag,_,done(_,removeRemains,Ag),_))
   : my_name(Ag) & account(errorCode(archaeologicalRemains))
  <- <remove remains very carefully>;
     goalAchieved(removeRemains);
     resetGoal(sitePrepared). //another attempt can be made
```

The first plan is triggered when the provided account denotes that the failure is due to a flooding. In this case, the treatment includes water removal in order to restore the site. Once the exception has been handled, the initial goal sitePrepared is marked as released through operation `goalReleased(...)`, provided by the extended organizational infrastructure. By doing so, the agent states that the failure has been solved and the house construction can proceed. The second plan, in turn, is activated if the failure is due to the finding of some archaeological remains. Here, the exception is handled by removing the remains and by resetting goal sitePrepared, so that another attempt can be made in site preparation.

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

engineering properties [33, 36, 44]. Low coupling is gained since agents dependence is limited to the exchange of an exception, specified as `Account Template` within the organization. High cohesiveness, instead, is obtained by ascribing the tasks of raising and handling exceptions to the agents that have the right functionalities to accomplish them.

7 Discussion and related works

Building upon suggestions from many works, we have defined a constructive technical framework of accountability (Definition 8) for supporting the realization of robust multiagent organizations, and we have illustrated how the framework can be mapped onto the JaCaMo agent platform and leveraged for practical agent programming. Accountability legitimates the a-taker to ask for an account, and creates the expectation that the a-giver will provide a meaningful account. The meaningfulness of the account is grounded on the *structural* dimension of accountability. This is the distinctive feature of accountability compared to other social relationships like business contracts and social commitments, which only yield obligations to do something; in other words, they present only a *normative* dimension.

The proposal features a clear separation of concerns: at design time the kinds of perturbation of interest are identified; how these are actually handled, however, depends on the specific plans (or behaviors) implemented by the agents (known only at runtime), playing organizational roles. Moreover, our proposal is useful to handle perturbations that cannot be properly addressed by a single agent. In many cases, the agent that detects a perturbation is not aware of the global context (e.g., how the perturbation may indirectly affect tasks of other agents), and has no power for fixing the problem. On the other hand, the agent that could handle the perturbation has no access to the situation where the perturbation has occurred. Accountability is the means through which an account about a perturbation is reported to the agent who is responsible for treating that perturbation. In some way, accountability complements the plan failure mechanism of JaCaMo (where failure is tackled by an agent locally), because it enables a sort of escalation of a failure.

The relevance of accountability in the design of agent organizations has been also highlighted in [22, 49], where the authors posit that *interaction* is the central notion around which a MAS should be designed. In particular, in their proposal agent interactions should only occur via social protocols, that specify social relationships via normative expectations (e.g., commitments, authorizations, prohibitions). Norms and organizations are, therefore, strictly related to each other: an organization is defined via norms, and a norm is defined in an organization [49]. Accountability emerges as an (implicit) directed relationship between agents, reflecting the legitimate expectations induced by the norms. Agents are, in fact, accountable for such expectations: they are free to violate them, but, then, they could be asked to account for their choice, and possibly be sanctioned. Contrary to our proposal, accountability in [22] is not explicitly modeled as a first-class notion. Rather, it is the consequence of the commitments taken on by the agents. Moreover, the structural dimension of accountability is not taken into account, thus there is no conceptual support in the definition of feedback chains that, as we have shown, are essential to convey sound accounts to the most competent agents in distributed scenarios.

Also our proposal takes for granted the existence of an organization as a context within which norms, including accountability, are defined. However, in [22, 49] the organization is just a set of norms, that specify the expected, correct behaviors of the agents. In our

case, instead, an organization is set up to achieve a complex goal; its purpose is to define a structure for distributing responsibilities among agents and allow the coordination of their activities. In this sense, the organization is a shared environment, which provides the agents artifacts for their coordination. The execution of tasks is carried out in a distributed manner by the agents, that play roles in the organization, and are autonomous and opaque to each other. The agents focus on organization artifacts, and in this way become aware of obligations, amounting to goals they should achieve. It is worth noting that the autonomy of the agents is preserved since, as in [22], agents are free to violate expectations by not discharging their obligations.

This work sets the ground for several future directions. First, it represents a general schema that can be tailored to specific applications, as illustrated in Sect. 6, e.g., to realize an exception handling mechanism in agent organizations, by constraining the way in which agents produce and consume accounts. The current proposal builds robustness on top of a given distribution of responsibility, but we mean to extend the study to the cases where responsibilities may change along time, either by effect of the received accounts or due to external circumstances; e.g., an agent leaving the organization or a bottleneck that is identified and solved by splitting some responsibilities among many agents. Another future direction of work concerns the realisation of tools that can support the work of actual organizations, like the mentioned UNDP, by combining accountability frameworks with oversight policies. This case is more general than the one we have tackled because accounts will often concern performances, and taking advantage of opportunities is also a concern.

In software engineering, robustness is considered a key property of software systems [40], and is usually gained by ensuring (at design time) that "exceptional" events will be reported to those software components which have the means for handling them properly. As pinned out in [45], traditional exception handling approaches, however, do not fit some key characteristics of multiagent systems, like openness, heterogeneity, agent encapsulation, and distribution. In particular, they usually assume that software components are collaborative, and that their code can be inspected while handling some given exception. But introspection is often impossible when dealing with agents, and collaboration cannot be given for granted. [45] suggests that an exception handling mechanism for multiagent systems should leverage both on the proactivity of agents, and on the environment in which agents are situated. Nevertheless, a few authors faced the problem of modeling exceptions in an agent-based system. Among them, [42] relies on commitment-based protocols, while [35] proposes an obligation-based approach for exception handling in interaction protocols. Some insight on how accountability and responsibility can support the realization of an exception handling mechanism in a multiagent environment can be found in [4, 5, 14].

In [21, 22] the authors explain how, within Socio-Technical Systems, accountability plays a fundamental role in balancing the principals' autonomy: a principal can decide to violate any expectation for which it is accountable, however, by way of accountability the principal would be held to account about that violation.

Accountability is recognized as a value for developing software also in [24], where a proposal complementary to ours is made. There, the authors focus on answer production in presence of an accountability relationship, tacking questions: how to properly define the temporal window to consider? Which pieces of information are relevant in this time interval? Which questions are suitable to be asked in this setting? The account giving agent produces an answer in terms of its internal mechanisms. The proposal, however, does not provide the organizational view of the system of interacting agents and does not tackle robustness and exceptions.

In [23], accountability enables the process of norms adaptation by feeding outcomes back into the design-phase. In this approach, the account is a justification of an agent's norm-violating behavior. This is a different understanding of accounts than ours because, for us, account givers are not rule violators: they meet perturbations, and provide information about the encountered situations. The account takers, on their hand, will interpret the received accounts – possibly combining them with further information provided by other agents or that simply belongs to the callee's level. The adaptation process in [23], that consists in norm modification, however, can be seen as a kind of robustness. Our objective is different: we do not target norm modification, but the achievement of the organizational goal despite the occurrence of perturbations. The two approaches are not in contrast, rather, they complement each other. They are both exemplifications of the perspective put forward in [1], for which a property of a system is robust if it is invariant with respect to a set of perturbations. The difference lies in the type of perturbations the two approaches aim at.

Finally, MOCA [9] provides an information model of accountability, that captures the kind of facts that must be available to allow the identification of account-givers in certain situation of interest. The model is given in Object-Role Modeling (ORM) [37] due to the relational nature of the represented concepts, and enables automatic verification of consistency The information model is centered around two basic concepts: *just expectation* and *control*. Just expectation is intended as the mutual awareness and acceptance of an accountability relationships between the involved a-giver and a-taker. Control, instead, is intended as the power, possibly exerted indirectly by means of other agents, of achieving a condition of interest. The normative and structural dimensions of accountability, that characterize our proposal, respectively capture these two features. Through the normative dimension, agents are aware of the obligations they may be subjected as a-givers, and what permissions they have as a-takers. The structural dimension, instead, grounds accountability relationships over an explicit assumption of responsibility by the agents, that we interpret as a declaration of direct control.

References

1. Alderson, D. L., & Doyle, J. C. (2010). Contrasting views of complexity and their implications for network-centric infrastructures. *IEEE Transactions Systems, Man, and Cybernetics - Part A: Systems and Humans, 40*(4), 839–852. https://doi.org/10.1109/TSMCA.2010.2048027.
2. Aldewereld, H., Boissier, O., Dignum, V., Noriega, P. & Padget, J. (eds.): (2016). Social coordination frameworks for social technical systems, *Law, Governance and Technology Series*, vol. 30. Springer International Publishing https://doi.org/10.1007/978-3-319-33570-4
3. Baldoni, M., Baroglio, C., Boissier, O., May, K.M., Micalizio, R. & Tedeschi, S. (2018)Accountability and responsibility in agent organizations. In T. Miller, N. Oren, Y. Sakurai, I. Noda, B.T.R. Savarimuthu, T. Cao Son (eds.) PRIMA 2018: Principles and Practice of Multi-Agent Systems - 21st International Conference, Tokyo, Japan, October 29 - November 2, Proceedings, *Lecture Notes in Computer Science*, vol. 11224, pp. 261–278. Springer (2018). https://doi.org/10.1007/978-3-030-03098-8_16
4. Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R. & Tedeschi, S. (2021) Demonstrating Exception Handling in JaCaMo. In F. Dignum, J.M. Corchado, F. De La Prieta (eds.) Advances in Practical Applications of Agents, Multi-Agent Systems, and Social Good. The PAAMS Collection - 19th International Conference, PAAMS Salamanca, Spain, October 6-8, 2021, Proceedings, *Lecture Notes in Computer Science*, vol. 12946, pp. 341–345. Springer (2021). https://doi.org/10.1007/978-3-030-85739-4_28
5. Baldoni, M., Baroglio, C., Boissier, O., Micalizio, R. & Tedeschi, S. (2021). Distributing Responsibilities for Exception Handling in JaCaMo. In U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.) Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '21, pp. 1752–1754. IFAAMAS https://doi.org/10.5555/3463952.3464226

6. Baldoni, M., Baroglio, C., Capuzzimati, F., & Micalizio, R. (2018). Type checking for protocol role enactments via commitments. *Journal of Autonomous Agents and Multi-Agent Systems, 32*(3), 349–386. https://doi.org/10.1007/s10458-018-9382-3.
7. Baldoni, M., Baroglio, C., May, K.M., Micalizio, R. & Tedeschi, S. (2016) Computational accountability. In Proc. of the AI*IA Workshop on Deep Understanding and Reasoning, *CEUR Workshop Proceedings*, vol. 1802, pp. 56–62. CEUR-WS.org
8. Baldoni, M., Baroglio, C., May, K.M., Micalizio, R. & Tedeschi, S. (2018). Computational Accountability in MAS Organizations with ADOPT. Applied Sciences **8**(4) https://doi.org/10.3390/app8040489
9. Baldoni, M., Baroglio, C., May, K. M., Micalizio, R., & Tedeschi, S. (2019). MOCA: An ORM MOdel for computational accountability. *Journal of Intelligenza Artificiale, 13*(1), 5–20. https://doi.org/10.3233/IA-180014.
10. Baldoni, M., Baroglio, C. & Micalizio, R. (2020). Fragility and Robustness in Multiagent Systems. In C. Baroglio, J.F. Hubner, M. Winikoff (eds.) Post-Proc. of the 8th International Workshop on Engineering Multi-Agent Systems, EMAS 2020, Revised Selected Papers, no. 12589 in LNAI, pp. 61–77. Springer, Auckland, New Zealand https://doi.org/10.1007/978-3-030-66534-0
11. Baldoni, M., Baroglio, C., Micalizio, R. & Tedeschi, S. (2020) Is explanation the real key factor for innovation? In: C. Musto, D. Magazzeni, S. Ruggieri, G. Semeraro (eds.) Proceedings of the Italian Workshop on Explainable Artificial Intelligence co-located with 19th International Conference of the Italian Association for Artificial Intelligence, XAI.it@AIxIA 2020, Online Event, November 25-26, *CEUR Workshop Proceedings*, vol. 2742, pp. 87–95. CEUR-WS.org http://ceur-ws.org/Vol-2742/short2.pdf
12. Baldoni, M., Baroglio, C., Micalizio, R. & Tedeschi, S. (2021). Reimagining robust distributed systems through accountable MAS. IEEE Internet Computing **25**(6) https://doi.org/10.1109/MIC.2021.3115450
13. Baldoni, M., Baroglio, C., Micalizio, R. & Tedeschi, S. (2021). Robustness based on Accountability in Multiagent Organizations. In U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.) Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '21, pp. 142–150. IFAAMAS https://doi.org/10.5555/3463952.3463975
14. Baldoni, M., Baroglio, C., Micalizio, R. & Tedeschi, S. (2022). Exception Handling as a Social Concern. IEEE Internet Computing, https://doi.org/10.1109/MIC.2021.3115450
15. Bauer, B., Müller, J., & Odell, J. (2001). Agent UML: A formalism for specifying multiagent software systems. *Software Engineering and Knowledge Engineering, 11*(3), 207–230. https://doi.org/10.1142/S0218194001000517
16. Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., & Santi, A. (2013). Multi-agent oriented programming with JaCaMo. *Science of Computer Programming, 78*(6), 747–761. https://doi.org/10.1016/j.scico.2011.10.004
17. Boissier, O., Bordini, R.H., Hübner, J.F. & Ricci, A. (2019). Dimensions in programming multi-agent systems. *The Knowledge Engineering Review,* **34** https://doi.org/10.1017/S026988891800005X
18. Bordini, R. H., Hübner, J. F., & Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason*. USA: John Wiley & Sons.
19. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems, 8*(3), 203–236. https://doi.org/10.1023/B:AGNT.0000018806.20944.ef
20. de Brito, M., Hübner, J. F., & Boissier, O. (2017). Situated artificial institutions: stability, consistency, and flexibility in the regulation of agent societies. *Autonomous Agents and Multi-Agent Systems,* 1–33. https://doi.org/10.1007/s10458-017-9379-3.
21. Chopra, A.K. & Singh, M.P. (2014). The thing itself speaks: Accountability as a foundation for requirements in sociotechnical systems. In IEEE 7th Int. Workshop RELAW. IEEE Computer Society https://doi.org/10.1109/RELAW.2014.6893477
22. Chopra, A.K. & Singh, M.P. (2016). From social machines to social protocols: Software engineering foundations for sociotechnical systems. In Proceedings of the 25th International Conference on World Wide Web, pp. 903–914 https://doi.org/10.1145/2872427.2883018
23. Chopra, A.K. & Singh, M.P. (2018). Sociotechnical Systems and Ethics in the Large. In: J. Furman, G.E. Marchant, H. Price, F. Rossi (eds.) Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society, AIES 2018, New Orleans, LA, USA, February 02-03, 2018, pp. 48–53. ACM https://doi.org/10.1145/3278721.3278740
24. Cranefield, S., Oren, N. & Vasconcelos, W.W. (2018). Accountability for practical reasoning agents. In: M. Lujak (ed.) Agreement Technologies - 6th International Conference, AT 2018, Bergen, Norway, December 6-7, 2018, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 11327, pp. 33–48. Springer https://doi.org/10.1007/978-3-030-17294-7_3

25. Dardenne, A., van Lamsweerde, A., & Fickas, S. (1993). Goal-directed requirements acquisition. *Science of Computer Programming, 20*(1), 3–50. https://doi.org/10.1016/0167-6423(93)90021-G
26. Dignum, V., Dignum, F., & Meyer, J. J. (2004). An agent-mediated approach to the support of knowledge sharing in organizations. *The Knowledge Engineering Review, 19*(2), 147–174. https://doi.org/10.1017/S0269888904000244
27. Dignum, V., Vázquez-Salceda, J. Dignum, F. (2004). OMNI: introducing social structure, norms and ontologies into agent organizations. In Programming Multi-Agent Systems, Second International Workshop ProMAS, Selected Revised and Invited Papers, *Lecture Notes in Computer Science*, vol. 3346, pp. 181–198. Springer https://doi.org/10.1007/978-3-540-32260-3_10
28. Dubnick, M.J. & Justice, J.B. (2004) Accounting for accountability. Annual Meeting of the American Political Science Association
29. Elder-Vass, D. (2011). *The Causal Power of Social Structures: Emergence, Structure and Agency*. Cambridge: Cambridge University Press.
30. Executive Board of the United Nations Development Programme and of the United Nations Population Fund: The UNDP accountability system, accountability framework and oversight policy. Tech. Rep. DP/2008/16/Rev.1, United Nations (2008)
31. Feltus, C. (2014) Aligning access rights to governance needs with the responsability metamodel (ReMMo) in the frame of enterprise architecture. Ph.D. thesis, University of Namur, Belgium
32. Garfinkel, H. (1967) Studies in ethnomethodology. Prentice-Hall Inc.
33. Goodenough, J. B. (1975). Exception handling: Issues and a proposed notation. *Communication ACM, 18*(12), 683–696. https://doi.org/10.1145/361227.361230.
34. Günay, A., Winikoff, M. & Yolum, P. (2012). Commitment protocol generation. In: M. Baldoni, L.A. Dennis, V. Mascardi, W.W. Vasconcelos (eds.) Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 7784, pp. 136–152. Springer https://doi.org/10.1007/978-3-642-37890-4_8
35. Gutierrez-Garcia, J.O., Koning, J. & Ramos-Corchado, F. (2009). An obligation approach for exception handling in interaction protocols. In 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology, vol. 3, pp. 497–500 https://doi.org/10.1109/WI-IAT.2009.334
36. Hägg, S. (1997). A sentinel approach to fault handling in multi-agent systems. In Multi-Agent Systems Methodologies and Applications, pp. 181–195. Springer Berlin Heidelberg
37. Halpin, T., & Morgan, T. (2008). *Information Modeling and Relational Databases*. USA: Morgan Kaufmann Publishers.
38. Hart, H. L. A. (1968). *Punishment and responsibility*. Oxford: The Clarendon Press.
39. Hubner, J. F., Sichman, J. S., & Boissier, O. (2007). Developing organised multiagent systems using the MOISE+ model: Programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering, 1*(3/4), 370–395. https://doi.org/10.1504/IJAOSE.2007.016266
40. ISO/IEC/IEEE: Systems and software engineering - Vocabulary (2017)
41. López y López, F. & Luck, M. (2003). Modelling norms for autonomous agents. In 4th Mexican International Conference on Computer Science (ENC 2003), 8-12 September 2003, Apizaco, Mexico, pp. 238–245. IEEE Computer Society https://doi.org/10.1109/ENC.2003.1232900
42. Mallya, A.U. & Singh, M.P. (2005). Modeling exceptions via commitment protocols. In Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '05, pp. 122–129. ACM https://doi.org/10.1145/1082473.1082492
43. Micalizio, R., & Torasso, P. (2014). Cooperative monitoring to diagnose multiagent plans. *Journal of Artificial Intelligence Research, 51*, 1–70. https://doi.org/10.1613/jair.4339
44. Miller, R., & Tripathi, A. (2004). The guardian model and primitives for exception handling in distributed systems. *IEEE Transactions on Software Engineering, 30*(12), 1008–1022. https://doi.org/10.1109/TSE.2004.106
45. Platon, E., Sabouret, N. & Honiden, S. (2007). Challenges for exception handling in multi-agent systems. In Software Engineering for Multi-Agent Systems V, pp. 41–56. Springer https://doi.org/10.1145/1138063.1138072
46. Ricci, A., Piunti, M., Viroli, M. & Omicini, A. (2009). Environment Programming in CArtAgO, pp. 259–288. Springer US https://doi.org/10.1007/978-0-387-89299-3_8
47. Romzek, B. S., & Dubnick, M. J. (1987). Accountability in the Public Sector: Lessons from the Challenger Tragedy. *Public Administration Review, 47*(3).
48. Singh, M.P. (2003). 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 https://doi.org/10.1145/860575.860721
49. Singh, M. P. (2013). Norms as a basis for governing sociotechnical systems. *ACM Transactions on Intelligent Systems and Technology, 5*(1), 21. https://doi.org/10.1145/2542182.2542203
50. Telang, P.R., Singh, M.P. & Yorke-Smith, N. (2011). Relating Goal and Commitment Semantics. In Programming Multi-Agent Systems - 9th Int. Workshop, ProMAS, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 7217, pp. 22–37. Springer https://doi.org/10.1007/978-3-642-31915-0_2
51. Telang, P. R., Singh, M. P., & Yorke-Smith, N. (2019). A coupled operational semantics for goals and commitments. *Journal of Artificial Intelligence Research, 65*, 31–85. https://doi.org/10.1613/jair.1.11494
52. Thangarajah, J., Harland, J., Morley, D.N. & Yorke-Smith, N. (2010). Operational behaviour for executing, suspending, and aborting goals in BDI agent systems. In A. Omicini, S. Sardiña, W.W. Vasconcelos (eds.) Declarative Agent Languages and Technologies VIII - 8th International Workshop, DALT 2010, Toronto, Canada, May 10, 2010, Revised, Selected and Invited Papers, *Lecture Notes in Computer Science*, vol. 6619, pp. 1–21. Springer https://doi.org/10.1007/978-3-642-20715-0_1
53. Winikoff, M., Padgham, L., Harland, J. & Thangarajah, J. (2002) Declarative & procedural goals in intelligent agent systems. In D. Fensel, F. Giunchiglia, D.L.M. Guinness, M. Williams (eds.) Proc. of the 8th Int. Conf. on Principles and Knowledge Representation and Reasoning (KR-02), pp. 470–481. Morgan Kaufmann
54. Zambonelli, F., Jennings, N. R., & Wooldridge, M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering Methodology, 12*(3), 317–370. https://doi.org/10.1145/958961.958963