# Towards Forward Responsibility in BDI Agents

Rafael C. Cardoso[1], Angelo Ferrando[2], Joe Collenette[3], Louise A. Dennis[3], and Michael Fisher[3]

[1] University of Aberdeen, Aberdeen, UK
rafael.cardoso@abdn.ac.uk
[2] University of Genova, Genova, Italy
angelo.ferrando@unige.it
[3] The University of Manchester, Manchester, UK
{joe.collenette,louise.dennis,michael.fisher}@manchester.ac.uk

**Abstract.** In this paper, we discuss forward responsibilities in Belief-Desire-Intention agents, that is, responsibilities that can drive future decision-making. We focus on individual rather than global notions of responsibility. Our contributions include: (a) extended operational semantics for responsibility-aware rational agents; (b) hierarchical responsibilities for improving intention selection based on the priorities (i.e., hierarchical level) of a responsibility; and (c) shared responsibilities which allow agents with the same responsibility to update their priority levels (and consequently commit or not to the responsibility) depending on the lack (or surplus) of agents that are currently engaged with it.

**Keywords:** forward-looking responsibility · task responsibility · BDI agents.

## 1 Introduction

A recent "Blue Sky Ideas" paper [26] discussed existing research and new research opportunities in the application of responsibility for trustworthy autonomous systems. They describe many research themes and challenges, but of particular interest to us is the challenge of using responsibility "to ensure system reliability and fault tolerance in the technical software development context".

We refer to rational agents as cognitive programmable entities that perform autonomous decision making by reasoning about events, capabilities, and knowledge of the world. Recent literature reviews on agent-oriented programming have highlighted the need for safer and more reliable agents [16, 6, 9].

In this paper we focus on *forward-looking* (as opposed to *backward-looking*) responsibilities [18]. In the context of rational agents, the former uses responsibilities to aid in the process of task selection, while the latter is related to the notions of accountability, liability, and blameworthiness. Many concepts of responsibility exist, see [26] for a more comprehensive discussion.

We illustrate these different concepts of responsibility in Figure 1. Besides the dimension regarding its meaning, when considering responsibility-aware agents

we also have to consider if the view of responsibility that was chosen is centralised or decentralised. A centralised view of responsibility in autonomous agents occurs when the information about responsibilities (either backward or forward-looking) is stored in a shared environment (e.g., organisation), which makes it easier for any agent to access the responsibilities of other agents. On the other hand, a decentralised view usually requires communication between agents in order to obtain access to other agents responsibilities. Whereas previous literature about responsibility in agent systems have often focussed on organisations or some other centralised model of responsibilities [25, 24, 2, 3], in this paper we focus on forward-looking responsibilities where our agents reason about their own individual responsibilities and have an individual (decentralised) view of responsibilities. We achieve this by extending well know formal theories for agent computational models.

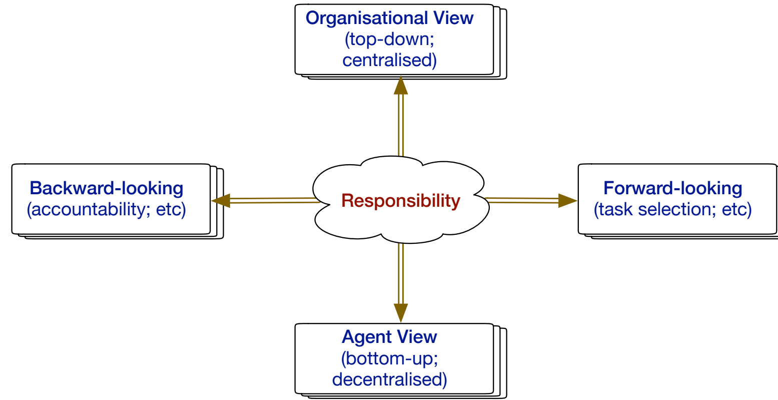


**Fig. 1.** Different dimensions when considering responsibility for rational agents. Directions of arrows and axes have no additional semantic meaning, an axis simply represents a different dimension.

Formal agent theories, such as those based on the Belief-Desire-Intention (BDI) model [8], do not include the notion of responsibility. Models where responsibilities (and the similar concept of agent roles) have been described predominantly take a centralised/organisational view. We propose extending agent models and theories with forward-looking, decentralised responsibility and instantiate this by considering their computation in the reasoning cycle of agents.

Our concept of forward-looking responsibilities is distinctively different from concepts of beliefs and organisational roles, and as such it would not be possible to flatten our representation to either of these concepts without losing some of our contributions and still maintain the original identity of beliefs and organisational roles. Most notably, one of these contributions is the direct impact that responsibilities (in particular their hierarchy) causes in the reasoning cycle by guiding the intention selection of rational agents.

## 2 Related Work

The work in [25] presents a strategic reasoning approach for tackling backward-looking responsibility in rational agents. They specify the system as a Concurrent Epistemic Game Structure (CEGS), and apply formal verification of strategic properties (Alternating-time Temporal Logic in particular) to conclude the responsibility of the agents w.r.t. the occurrence of some bad event (e.g., applied to an example where some agents want to poison a certain agent, and when the agent dies, they want to know who was responsible for it, and in which amount). In further extension of their work a task coordination framework is proposed, TasCore [24], which is a dynamic task coordination method for multi-agent strategic reasoning. There are two main parts of TasCore: task allocation and a retrospective mechanism for ascribing responsibilities to agents. The latter is based on their previous work of assigning degrees of responsibilities based on past history, which in this case relates to the tasks that have been allocated and how they have been fulfilled. Both works are based on the notion of backward-looking responsibilities. This paper explores forward-looking responsibilities, where agents adopt responsibilities and tasks are then attributed to the agent.

A series of research papers have tackled the notion of accountability in multi-agent organisations as a means to improve robustness of the system [2, 1, 3]. In these papers, the definition of responsibility is closely related to that of roles in multi-agent organisations. That is, a responsibility is a collection of tasks that should be performed within a society of agents. Agents are assumed to be autonomous, and therefore must explicitly commit to responsibilities that they want oversee. Accountability is represented through accountability agreements between a pair of agents, where one agent can ask for an account about a particular task to the other agent. Robustness is obtained by connecting failed accounts to recovery strategies, which in turn can trigger treatment tasks that eventually lead to new commits to responsibilities. The main difference between their work and ours is that they take an organisational view of responsibilities rather than our individual agent view.

Other works that are tangential to ours include: the missions in the organisational layer of the JaCaMo multi-agent programming framework [5] that resemble our notion of forward-looking responsibility in that they also serve as triggers for adding a collection of goals, but provide no means of automatically reasoning about them, and their use requires a centralised organisation; maintenance goals [14] seek to maintain a particular state of the world, which share some similarities with our work, but in our case we are more interested in the overall behaviour of the agent and its impact in intention selection; and a set of requirements for accountability in autonomous agents [11] with a strong focus on organisational norms. However, none of the above consider the concept of forward-looking responsibility reasoning in rational agents with an individual agent view.

## 3 Responsibility-Aware Agents

We formalise responsibility by extending the syntax and operational semantics of BDI agent languages such as AgentSpeak(L) [20], AgentSpeak$^{-}$ [12], and Jason [7]. Note that we do not present the complete syntax or formal semantics for these languages, we simply report the necessary rules and extensions for obtaining responsibility-aware agents. Further considerations may be necessary when trying to implement it in these (and other) agent languages. Nonetheless, the observations about the implementation details that we provide in Section 4 may be of some help.

We focus on BDI because it is the most traditional implementation of rational agents with a rich selection of programming languages (Jason, JaCaMo [4], ASTRA [10], CAN [23], Gwendolen [13], etc.) and has been shown in recent surveys [16, 6, 9] to have many open research problems that still need to be solved.

The BDI model of agency [8, 19] revolves around three main attitudes: *beliefs* is the knowledge that the agent has about the world; *desires* are goals that the agent wants to achieve; and *intentions* are the means of achieving the goals that the agent has committed to. BDI agents have a reasoning cycle that follows the 'sense-plan-act' methodology. The sense phase consists of receiving perceptions from the environment and messages from other agents. The plan phase starts with the generation of events, which can come from the addition/deletion of beliefs or goals. These events trigger the plan selection mechanism, which consults the plan library for relevant and applicable plans and then selects one to be added to the intention stack. The act phase removes and executes the top intention in the selected intention stack.

We add the notion of task responsibility (henceforth referred to simply as responsibility) in the reasoning cycle of the agent. Informally, responsibility is a task containing a collection of goals that relate to an overarching topic (e.g., responsibility for safety). When an agent adopts a responsibility, an event is generated that triggers the associated plan to start pursuing the goals that it is now responsible for achieving. An overview of the resulting reasoning cycle for responsibility-aware agents is shown in Figure 2.

We show the syntax for rational agents with responsibilities in Figure 3. The differences from the traditional syntax found in most BDI-based agent languages are the addition of a responsibility base along with the notion of responsibilities and the respective triggering events from adopting/dropping responsibilities, as well as adding support for updating the responsibility base inside the body of a plan. Note that the dynamic creation of responsibilities is not supported in this paper, the update simply refers to adopting or dropping a responsibility. We also support expressing responsibilities in the context of a plan, even thought this is not used directly in our theory, it can be useful in practice when building an agent program.

The responsibility base is there to provide a clear separation from the belief base. Each agent has its own individual responsibility base. An agent is capable of handling any responsibility in its base, but it does not initially commit to any

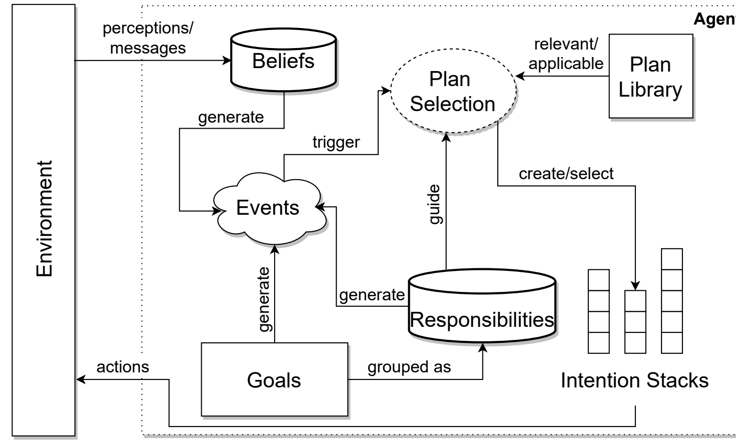


**Fig. 2.** Responsibility-aware BDI agent reasoning cycle.

of them by default. Adopting and dropping[4] responsibilities have to be manually inserted in the agent program, since the best moments at which to do this will require domain specific information. A responsibility that is dropped remains in the responsibility base because the agent can decide that it needs to adopt it again in the future.

A key difference between responsibility and belief bases is that responsibilities can only be defined at design time. Nevertheless, the following changes can occur to them at runtime: an agent can decide to adopt or drop a responsibility, thus altering the number of agents currently committed to it (i.e., active agents); and the priority that the agent has for a responsibility can be changed depending on specific circumstances at runtime related to shared responsibilities. Priorities and how they are used to guide intention selection are presented in Section 3.1. Shared responsibilities and how they can alter the priority that an agent has for a responsibility through agent communication are covered in Section 3.2.

To illustrate a typical responsibility, let us consider an example where a domestic robot is embedded with a rational agent that performs the high-level decision making. This agent has the following responsibility in its responsibility base:

$$cleaning([clean(bathroom), clean(bedroom)], 0, 1)$$

where the responsibility name is cleaning, the associated goals are to clean the bathroom and the bedroom, the current number of active agents committed to this responsibility is 0, and the recommended number is 1. Because we only have one responsibility in this example we omit the hierarchy (this is discussed in Section 3.1).

[4] If the goals associated with the responsibility have been successfully achieved, then the responsibility is automatically dropped.

```
agent     ::= bb  rb  pl
bb        ::= belief1 ... beliefn              (n ≥ 0)
rb        ::= resp1 ... respn  h               (n ≥ 0)
pl        ::= plan1 ... plann                  (n ≥ 1)
belief    ::= at
g         ::= at
at        ::= P(t1,...,tn)                     (n ≥ 0)
resp      ::= P([g1,...,gn],na,rec)            (n ≥ 1)
h         ::= hierarchy([hl1 ... hln])         (n ≥ 1)
hl        ::= [Presp1 ... Prespn]              (n ≥ 1)
plan      ::= te : {context} ← body
te        ::= +!g     | +belief   | −belief
              |  +/resp   |  −/resp
context   ::= ct1     |  ⊤
ct1       ::= belief  |  ¬belief  |  resp
              |   ¬resp  |  ct1∧ct1
body      ::= bd1,⊤   |  ⊤
bd1       ::= +!g     |  action   |  bbupdate
              |  rbupdate  |  bd1;bd1
action    ::= A(t1,...,tn)                     (n ≥ 0)
bbupdate  ::= +belief  |  −belief
rbupdate  ::= +/resp   |  −/resp
```

**Fig. 3.** Syntax for responsibility-aware rational agents. `bb` is belief base. `rb` is responsibility base. `pl` is plan library. `g` is goal. `at` is an atomic formulae with `P` as a predicate name and ($t_1, \dots, t_n$) as first-order logic terms. `resp` is responsibility with `na` as the current number of "active" agents committed to this responsibility and `rec` as the recommended number of agents. `h` is the hierarchy of responsibilities, it uses the reserved word (and terminal symbol) `hierarchy` as the predicate name. `hl` is a hierarchical level containing a partial order of responsibilities in that level. `te` is triggering event. `ct1` and `bd1` are context and body (resp.) to support chaining. `A` in `action` is a predicate name for the action. +/`resp` and −/`resp` have the extra forward slash symbol to differentiate it from belief operations.

Each responsibility also has a corresponding plan that triggers once the agent has decided to adopt the responsibility. For the previous example we would have the following plan:

```
+/cleaning : {⊤}
  ← +!clean(bathroom),
    +!clean(bedroom),
    −/cleaning.
```

The context of the plan is always true and the body of the plan contains the goals associated with the responsibility. Note that the body of plans usually follow a sequential composition, which means that the order in which the goals appear here is the order that they will be attempted to be achieved. At the end of the plan the cleaning responsibility is dropped.

The syntax from Figure 3 does not cover plan selection or the way that intention stacks work since these are not elements that can be expressed by the user of the language. Instead, these elements are controlled internally by rules and functions in the language. Next, we describe the standard operational semantics for these rules, since they are required to introduce our new extensions for adopting and dropping a responsibility, as well as to better understand where the contributions that are presented later on fit in the reasoning cycle of the agent. Due to space constraints we omit the cases for most of the rules where we would need an additional rule for when there are no elements to consume/handle, in which case the reasoning cycle would simply skip to another phase of the reasoning cycle. For example, rules that deal with empty elements in plan selection would skip to the intention selection phase. To improve readability we also omit the use of unifiers.

The inference rules that define the operational semantics represent transitions between agent configurations in the reasoning cycle of an agent. An agent configuration is denoted as:

$$Conf = \langle agent, C, M, T, rule \rangle$$

where *agent* is the agent program composed of a belief base, a responsibility base, and a plan library; $C$ is an agent's current circumstance represented by the tuple $\langle I, E, A \rangle$, respectively the set of intention stacks (sometimes referred to as intended means), set of events, and set of actions; $M$ represents the asynchronous communication between agents as a tuple $\langle In, Out \rangle$, respectively the mail box and outgoing messages to be sent; $T$ is an auxiliary structure that stores relevant temporary information that can be useful within a cycle, it is a tuple $\langle Rel, App, ev, ie, si, res, pl \rangle$ with $Rel$ the set of relevant plans, $App$ the set of applicable plans, $ev$, $ie$, $si$, $res$, and $p$ a particular event, intention associated with event, intention selected for execution, responsibility, and plan (respectively); and *rule* is the current step in the agent's reasoning cycle, representing which inference rule will be used in that step. To refer to sub-elements of an element in a tuple, such as the set of intentions in a circumstance we use $C_I$, similarly, $C_E$ for set of events $C_A$ for set of actions, and so on.

Plan selection is often separated into four phases: (1) selection of an event; (2) obtaining relevant plans; (3) obtaining applicable plans; and (4) selection of a plan.

*Selection of an event.* We need to select an event from the events that are currently active. The following inference rule is used for selecting an event[5]:

$$\text{(SelEv)}\frac{SelectEvent(C_E) = te}{\langle agent, C, M, T, SelEv \rangle \rightarrow \langle agent, C', M, T', RelPl \rangle}$$

$$\begin{aligned} where \quad & C'_E = C_E \setminus \{te\} \\ & T'_{ev} = te \\ & T'_{ie} = GetIntention(te) \end{aligned}$$

[5] In subsequent inference rules we assume that elements of the state remain unchanged unless explicitly stated in the rule.

This rule says that the *SelectedEvent* function uses the event set in the circumstance $C_E$ to select a triggering event $te$. Usually implementations of this function will simply select an event following the ordering method first in, first out. The rule updates the event set in the circumstance by removing the selected event from it, as well as assigning the event to the respective auxiliary structure to be used in further rules. The get intention function returns the intention associated with the selected event if that event was generated from previous executions of other plans (i.e., internal event), or $\top$ if it was generated from a perception (i.e., external event).

*Obtaining relevant plans.* It is necessary to obtain the relevant plans that match the selected event, which can be obtained with the rule:

$$\text{(RelPl)}\frac{RelevantPlans(T_e) \neq \varnothing}{\langle agent, C, M, T, RelPl\rangle \rightarrow \langle agent, C, M, T', AppPl\rangle}$$

$$where \quad T'_{Rel} = RelevantPlans(T_e)$$

The relevant plans function is straightforward since we simply have to match the previously selected event with the triggering event of plans in the plan library.

*Obtaining applicable plans.* To obtain the applicable plans we need to compare the set of relevant plans to the belief base:

$$\text{(AppPl)}\frac{ApplicablePlans(agent_{bb}, T_{Rel}) \neq \varnothing}{\langle agent, C, M, T, AppPl\rangle \rightarrow \langle agent, C, M, T', SelPl\rangle}$$

$$where \quad T'_{App} = ApplicablePlans(agent_{bb}, T_{Rel})$$

The applicable plans function will iterate over each relevant plan while checking the context of the plan against the belief base. If the result of this check is true, then the plan is applicable.

*Selection of a plan.* A plan is selected for execution from the applicable plans:

$$\text{(SelPl)}\frac{SelectApplicable(T'_{App}) = plan}{\langle agent, C, M, T, SelPl\rangle \rightarrow \langle agent, C, M, T', UpdtSt\rangle}$$

$$where \quad T'_{pl} = plan$$

The function for selecting an applicable plan usually picks the first in a top to bottom order from where they appear in the plan library.

Once a plan is selected, then a new intention stack for the instance of that plan is created if the intention associated with the triggering event is external:

$$\text{(CrtSt)}\frac{T_{ie} = \top \wedge T_{pl} = plan}{\langle agent, C, M, T, CrtSt\rangle \rightarrow \langle agent, C', M, T, SelInt\rangle}$$

$$where \quad C'_I = C_I \cup \{plan\}$$

Otherwise, with an internal event we use rule *UpdtSt* to update an existing stack of intentions:

$$(\text{UpdtSt})\frac{T_{ie} = intention \land T_{pl} = plan}{\langle agent, C, M, T, UpdtSt\rangle \rightarrow \langle agent, C', M, T, SelInt\rangle}$$

$$where \quad C'_I = (C_I \setminus \{intention\}) \cup \{intention[plan]\}$$

In this case, we update the intention by adding the selected plan at the bottom of the associated intention stack (represented by *intention[plan]*).

Plans can be instantiated into separate intention stacks, i.e., at any moment we can have more than one intention stack. Therefore, it is necessary to have a way to select the next intention stack to be executed:

$$(\text{SelInt})\frac{C_I \neq \varnothing \land SelectIntention(C_I) = i}{\langle agent, C, M, T, SelInt\rangle \rightarrow \langle agent, C, M, T', ExecInt\rangle}$$

$$where \quad T'_{si} = i$$

The implementation of the select intention function usually attempts to select an intention stack based on fairness to avoid starvation. For now we assume a similar behaviour, but in Section 3.1 we discuss a different implementation that will instead prioritise intention stacks related to responsibilities based on the existing partial order in the hierarchy.

In practice, an intention stack is composed of the bodies of selected plans (or subplans), and as such can include the following (see body in the syntax from Figure 3 for a complete list): an action, a belief update, a new goal, etc. Each of them will have their own individual rules that are activated when *ExecInt* is called. For the sake of brevity we only show the new operational semantics inference rules for executing intentions that are related to updating the responsibility base (i.e., adopting or dropping a responsibility), the remaining rules are all unchanged and thus similar to past work [20, 12, 7, 22].

If the head (topmost) intention in the selected intention stack is the adoption of a responsibility, then the following rule applies:

$$(\text{AResp})\frac{T_{si} = i[head \leftarrow +/resp; body]}{\langle agent, C, M, T, AResp\rangle \rightarrow \langle agent', C', M, T', ClrInt\rangle}$$

$$\begin{aligned} where \quad & agent'_{rb} = (agent_{rb} \setminus resp) \cup UpdateAdopt(resp) \\ & C'_E = C_E \cup \{+/resp\} \\ & C'_I = C_I \setminus \{T_{si}\} \\ & T'_{res} = resp \end{aligned}$$

The function to update a responsibility after it has been adopted is used to update the number of agents currently committed to it (in this case, increasing it by 1). More importantly, the set of events is updated to include a new event about this responsibility, which will eventually trigger (when the event is selected) the corresponding plan containing the list of goals related to it. Lastly, the selected intention is removed from the set of intentions.

Otherwise, if the head is the dropping of a responsibility:

$$\text{(DResp)}\ \frac{T_{si} = i[head \leftarrow -/resp; body]}{\langle agent, C, M, T, DResp\rangle \rightarrow \langle agent', C', M, T', DropInt\rangle}$$

$$\begin{array}{ll} where & agent'_{rb} = (agent_{rb} \setminus resp) \cup UpdateDrop(resp) \\ & C'_E = C_E \cup \{-/resp\} \\ & C'_I = C_I \setminus \{T_{si}\} \\ & T'_{res} = resp \end{array}$$

In this case, the function to update a responsibility decreases the number of agents currently committed to it by 1, and the selected intention that was just processed is removed from the set of intentions. Additionally, this rule leads to the rules for dropping an intention about a responsibility. Rule $DInt1$ is used when the drop is invoked from the body of the responsibility plan:

$$\text{(DInt1)}\ \frac{T_{res} \subseteq T_{pl}}{\langle agent, C, M, T, DInt1\rangle \rightarrow \langle agent, C, M, T, ClrInt\rangle}$$

This means that the responsibility is dropped because it came to a natural conclusion and there are no additional operations to make.

Rule $DInt2$ deals with the opposite condition:

$$\text{(DInt2)}\ \frac{T_{res} \nsubseteq T_{pl}}{\langle agent, C, M, T, DInt2\rangle \rightarrow \langle agent', C', M, T', ClrInt\rangle}$$

$$where \quad C'_I = C_I \setminus DropIntention(T_{res})$$

This means that the responsibility is dropped from a plan outside the original plan, which means that the agent has autonomously decided to stop being responsible for it (e.g., something has failed, or another responsibility with a higher priority that conflicts with this one has been adopted). The drop intention function drops the intention associated with the responsibility, which in this case will drop the corresponding plan along with its subplans (plans for the goals listed within the responsibility).

Rules *AResp*, *DInt1*, and *DInt2* lead to the *ClrInt* rule, which we omit because it simply removes empty stacks of intentions and then proceeds to the beginning of a new cycle.

### 3.1 Priorities and Hierarchy of Responsibilities

Each agent has it own individual responsibility base which includes not only the responsibilities that the agent can adopt but also a hierarchy that determines the priority of a responsibility in relation to others by categorising them into different hierarchical levels. Responsibilities and the hierarchy are defined at design time and our theory does not (yet) provide support for them to be changed dynamically at execution time. The only exceptions are when the current number of agents committed to a responsibility changes (as shown in the rules *AResp* and

*DResp* with the *UpdateAdopt* and *UpdateDrop* functions), and when an agent tries to adopt a shared responsibility which can cause the hierarchy to change (this defined later in Section 3.2).

Recall function `SelectIntention`($C_I$) from rule *SelInt*; we now provide a pseudo-code implementation for it in Algorithm 1 where an intention stack is selected based on the priority defined by the hierarchical levels in the agent's hierarchy.

**Algorithm 1:** Selects an intention stack given a set of intentions stacks as input ($C_I$).

```
1  Function SelectIntention(C_I)
2      i ← ∅;
3      if C_I ≠ ∅ then
4          RespStacks ← GetRespStacks(C_I);
5          hlevel ← 0;
6          while there exists {stack} ∈ RespStacks do
7              slevel ← GetHLevel(stack);
8              if slevel > hlevel then
9                  hlevel ← slevel;
10                 i ← stack;
11             RespStacks ← RespStacks \ {stack};
12         if i = ∅ then
13             i ← SelectIntentionFairness(C_I);
14     return i
```

First, we initiate the return variable with null. Next, we check if the set of intention stacks is not empty, i.e., there is at least one active intention stack. If that is the case, then we use the function $GetRespStacks(C_I)$ from line 4 to get all intention stacks that contain a responsibility (we then call these the responsibility stacks). Because each responsibility has a corresponding plan, then whenever that plan is selected a new intention stack is created, and any subplans (e.g., from the associated goals) originated from it are attached to the same stack. We initiate the variable that holds the most prioritised hierarchical level (*hlevel*) with 0. Hierarchical levels are implicit within the hierarchy, and as such can be extracted with an appropriate function. The last hierarchical level starts at 1, and increases by 1 as it goes up the levels of the hierarchy.

The *while* loop (lines 6–11) iterates over each element in the set of responsibility stacks. In this loop we first get the hierarchical level of the responsibility attached to the stack (stored in variable *slevel*), and then check if that value is greater than our currently most prioritised hierarchical level. If it is then we update the related variables accordingly. Note that we do not test if the level is equal, since there is no priority between responsibilities within the same hier-

archical level. Thus, it will simply pick the stack with the responsibility that it processed first. At the end of the loop we remove the stack that was processed from the set of responsibility stacks.

It is possible for the set of intention stacks to not be empty while the set of responsibility stacks is. In this case, the if condition on lines 12–13 is triggered, which calls an intention stack selection function based on fairness. Our view of responsibilities in this work is that they should be intentionally prioritised over intentions belonging to other plans. This may lead to starvation of plans, especially any plans not related to a responsibility (such as plans for belief updates that come from the environment or other agents). An efficient implementation of our function requires to incorporate some level of fairness to attempt to avoid starvation while still preserving priority of hierarchical levels of responsibilities as much as possible. Fairness and starvation of threads/processes/resources are extensively researched topics in Software Engineering, and as such we do not tackle these concepts here.

To illustrate the use of the hierarchy of responsibilities, let us expand the previous example of a domestic robot by adding a few extra responsibilities and building a hierarchy:

```
cleaning([clean(bathroom),clean(bedroom)],0,1)
safety([locks(frontdoor),search(triphazards)],0,2)
cooking([cook(breakfast),makelist(grocery)],0,1)

hierarchy([[safety],[cleaning,cooking]])
```

Note that the responsibility base above is of a single agent, other agents in the system may have different configurations in their responsibility bases. In this extended example, we now have three responsibilities: cleaning, safety, and cooking. Safety is recommended to have up to two agents being responsible for it, while the others remain at only one. We provide a visual representation of the hierarchy for this example in Figure 4.

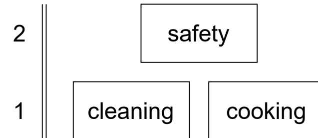


**Fig. 4.** Visual representation of a hierarchy.

The hierarchy has two levels, the bottom starts at 1 and contains the cleaning and cooking responsibilities (recall that responsibilities within the same hierarchical level have no relation of priority between each other), then the level above is 2 and contains safety. Effectively this means that safety has priority over (i.e., its hierarchical level is greater than) both cleaning and cooking. Note that the hierarchical levels do not need to be represented as numbers, for example we

could also use "Low Importance", "Medium Importance", "High Importance" by limiting the hierarchy to three levels (there is always one special level called "idle" which we present in the next section). In this visual representation and in our algorithm we used $\mathbb{N}^*$ to represent each level with larger numbers being higher up in the hierarchy. This is an implementation abstraction, and the information can be easily extracted from the hierarchy definition and transformed as preferred (some necessary small updates would need to be made to Algorithm 1 if not using numerical values).

### 3.2 Improving Reliability with Shared Responsibilities

An individual view of responsibilities requires some form of communication between the agents in order to keep track of shared responsibilities. Shared responsibilities are responsibilities that appear in more than one agent's responsibility base. Our mechanism to improve reliability in shared responsibilities is based around the recommended number of agents to commit to a responsibility. Once that number is met, then any agents that try to commit to it will become "backup" agents in the sense that they alter their responsibility hierarchy to place this responsibility in a special hierarchical level which we call "idle". If at any point this situation reverses (i.e., the number of agents committed to a responsibility drops below the recommended) then the backup agent has to revert the position of the responsibility in its hierarchy back to the original one.

We use a simple speech-act *tell(adopt(resp))* or *tell(drop(resp))* for sending information about a responsibility that the sender has in its own responsibility base to all other agents (i.e., a broadcast). An extensive discussion on the formal semantics of speech-act communication in BDI agent languages is available in [22].

We need to update some of the rules from previous sections to now account for communication and the reliability mechanism for shared responsibilities. In particular, the *AResp* rule presented previously has to be split into two. First, is the standard case of when the recommended number has not been achieved yet:

$$\text{(AResp1)}\,\frac{T_{si} = i[head \leftarrow +/resp; body] \wedge resp_{na} < resp_{rec}}{\langle agent, C, M, T, AResp1\rangle \rightarrow \langle agent', C', M', T', ClrInt\rangle}$$

$$\begin{array}{ll} where & agent'_{rb} = (agent_{rb} \setminus resp) \cup UpdateAdopt(resp) \\ & C'_E = C_E \cup \{+/resp\} \\ & C'_I = C_I \setminus \{T_{si}\} \\ & T'_{res} = resp \\ & M'_{Out} = M_{Out} \cup tell(adopt(resp)) \end{array}$$

Here we simply add an extra condition to our premise in the original rule to check that the number of agents committed to the responsibility ($resp_{na}$) is less than the number of agents recommended ($resp_{rec}$). Additionally, since we are now also concerned about updating other agents' information about the number of agents committed to a responsibility we add the performative *tell(adopt(resp))* to the outgoing messages of the agent. Upon receiving such a message, if the

agent has the responsibility mentioned in the message, then the agent calls the *UpdateAdopt(resp)* function to update the number of agents that are committed to it.

Otherwise, if the recommended number of agents has already been met then this agent becomes a "backup" agent:

$$\text{(AResp2)}\frac{T_{si} = i[head \leftarrow +/resp; body] \wedge resp_{na} \geq resp_{rec}}{\langle agent, C, M, T, AResp2\rangle \rightarrow \langle agent', C', M, T', ClrInt\rangle}$$

$$\begin{aligned} where \quad & C'_E = C_E \cup \{+/resp\} \\ & C'_I = C_I \setminus \{T_{si}\} \\ & T'_{res} = resp \\ & agent'_{rb[hierarchy]} = UpdtH(agent_{rb[hierarchy]}, resp) \end{aligned}$$

Apart from the addition of the extra condition in the premise, this rule also modifies the agent's hierarchy. Because the agent has adopted the responsibility in a "backup" capacity, it calls the function $UpdtH(agent_{rb[hierarchy]}, resp)$ which changes the hierarchical level of ($resp$) to idle. Note here that because this is a backup agent, we no longer update the number of active agents nor do we use the communication performative that are present in the previous rule. These two things are only done when the agent changes from backup to active, which is explained later on.

Similarly, we also have to update *DResp*, but in this case we only need one rule so we simply overwrite the previous one:

$$\text{(DResp)}\frac{T_{si} = i[head \leftarrow -/resp; body]}{\langle agent, C, M, T, DResp\rangle \rightarrow \langle agent', C', M', T', DropInt\rangle}$$

$$\begin{aligned} where \quad & agent'_{rb} = (agent_{rb} \setminus resp) \cup UpdateDrop(resp) \\ & C'_E = C_E \cup \{-/resp\} \\ & C'_I = C_I \setminus \{T_{si}\} \\ & T'_{res} = resp \\ & M'_{Out} = M_{Out} \cup tell(drop(resp)) \end{aligned}$$

The only difference here is the addition of the communication performative *tell* which will broadcast the message that an agent has dropped a responsibility, and therefore any agent that has the same responsibility will use this as a triggering event to call the *UpdateDrop(resp)* function, which works the same as before and simply decreases the number of agents currently committed to a responsibility.

Due to space constraints, we do not show the rules for processing the agent's outgoing messages or for processing incoming messages in the mail box, and instead refer to the work in [22] for a complete list. A couple of straightforward extensions are required in particular for processing incoming messages about responsibilities:

*tell(adopt(resp)) message*

- call function *UpdateAdopt(resp)* to increase by 1 the number of agents committed to the responsibility (only if present in the responsibility base).

*tell(drop(resp)) message*

- call function $\mathit{UpdateDrop}(resp)$ to decrease by 1 the number of agents committed to the responsibility (only if present in the responsibility base);
- if the result of the $\mathit{UpdateDrop}(resp)$ function causes the number of commitments to drop below the recommended number, then call the function $\mathit{RestoreH}(agent_{rb[hierarchy]}, resp)$ to restore the responsibility to its previous hierarchical level (only if it had been changed to idle in the first place);
- call function $\mathit{UpdateAdopt}(resp)$ to reflect that the backup agent has now become active; and
- send the broadcast $\mathit{tell}(\mathit{adopt}(resp))$ so that other agents update their information accordingly.

We also need to add support for discarding intention stacks for idle responsibilities in Algorithm 1. This is straightforward as we can simply extend the function $\mathit{GetRespStacks}(C_I)$ to return only non-idle responsibility stacks.

Note that we do not explicitly deal with coordination of shared responsibilities. By default, multiple agents adopting the same responsibility will perform all the goals associated with that responsibility. In practice, this could be solved in various different ways, such as through communication, organisations, argumentation, task allocation, etc. In this paper we are concerned with providing the basis for reasoning in responsibility-aware agents, which allow for these extensions to be developed in future work.

Furthermore, we define how the agent can drop a responsibility and what happens when it is dropped, and not the specification of when the agent should drop the responsibility (apart of course from when it believes its responsibility has come to a natural end) or why (e.g., non-conformance due to time-sensitive deadline or failure). Deciding when to drop a responsibility, and the reasoning behind the decision, is specific to the domain the agent is implemented in. Therefore being out of scope of this paper.

## 4 Towards Implementation

The agent languages that are based on AgentSpeak(L) are natural candidates for incorporating our extensions, especially those that have not extensively modified the original AgentSpeak(L) semantics. Two options that fit well in this category are Jason [7] and Gwendolen [13]. Other languages may be viable alternatives, but may also require additional implementation considerations. We limit this discussion to be about the most difficult challenges in implementation.

To implement our selection intention stack function from Algorithm 1 is relatively straightforward. In the Jason language this can be altered in the `TransitionSystem` class which represents most of the agent's reasoning cycle; in particular function `selectIntention` (part of the `Agent` class) which is called from the `applySelInt` method. Similarly, in the Gwendolen language this could be done in the `selectIntention` from the `AILAgent` class by extending the `SelectIntentionHeuristic` interface. The most challenging part is trying to

incorporate some level of fairness in order to prioritise the hierarchy of responsibilities while still avoiding starvation of intention stacks. There are different ways of tackling this scheduling problem, and one option would be to look at this as a multi-resource allocation problem (e.g., the work in [15]) where different types of stacks (intended means created from responsibilities, environment reaction, proactive goals, etc.) can be seen as different types of resources.

Another important aspect is the communication between agents that is required for keeping the agents' responsibility bases up to date. This is particularly important if in the application domain the number of recommended agents for a responsibility is strict (i.e., it has maximum limit). A direct implementation of our approach requires some level of centralisation about updates to agents responsibility bases or a period of negotiation/argumentation to ensure that the information is consistent across all agents. This is especially important when backup agents receive a message about a responsibility being dropped, since otherwise without some synchronous behaviour it can be possible that multiple backup agents will become active at the same time (same time here refers to the period of time before these agents receive a message from each other updating the number of active agents). Again there are many ways to solve this which depends on the constraints of the application domain, but for example one possibility is to add a centralised shared list of active agents for each responsibility as well as a backup list, and then consume from the backup list in FIFO order.

## 5 Conclusion

In this paper we have extended the traditional operational semantics of rational agents to include an individual agent view of forward-looking responsibilities. This improves the reliability of responsibility-aware agents on two fronts: (i) introducing a hierarchy of responsibilities allows us to reason about the partial order relation by extending the intention stack selection function to prioritise more important responsibilities, which leads to improving the reliability of the system; and (ii) adding the notion of shared responsibilities which is realised through agent communication and it is used to coordinate agents so that if an active agent drops a responsibility and as a result the number of recommended agents is not met, then one of the backup agents will become active.

There are many different ways of extending our approach in future work. In this paper we focussed on a rather explicit definition of how rational agents reason about responsibilities, but much freedom is left to the user of the language (e.g., when to adopt a responsibility); it could be interesting to investigate reasoning about responsibility at a more meta level in regards to how responsibilities relate to each other, such as conflicts; perhaps exploring recent advances in argumentation for agents [21, 17]. Such feature would also better justify adding support for the dynamic creation/deletion of responsibilities at runtime, which we did not consider for this paper. An efficient implementation in existing agent-based programming languages would also serve to better demonstrate the usefulness of our approach in practice.

## References


1. Baldoni, M., Baroglio, C., Micalizio, R.: Fragility and robustness in multiagent systems. In: Baroglio, C., Hubner, J.F., Winikoff, M. (eds.) Engineering Multi-Agent Systems. pp. 61–77. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-66534-0_4
2. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Implementing business processes in jacamo+ by exploiting accountability and responsibility. In: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems. p. 2330–2332. AAMAS '19, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2019)
3. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Robustness based on accountability in multiagent organizations. In: Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems. p. 142–150. AAMAS '21, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2021)
4. Boissier, O., Bordini, R., Hubner, J., Ricci, A.: Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo. Intelligent Robotics and Autonomous Agents series, MIT Press (2020)
5. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming **78**(6), 747–761 (Jun 2013). https://doi.org/10.1016/j.scico.2011.10.004
6. Bordini, R.H., Seghrouchni, A.E.F., Hindriks, K.V., Logan, B., Ricci, A.: Agent programming in the cognitive era. Auton. Agents Multi Agent Syst. **34**(2), 37 (2020). https://doi.org/10.1007/s10458-020-09453-y, https://doi.org/10.1007/s10458-020-09453-y
7. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
8. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press (1987)
9. Cardoso, R.C., Ferrando, A.: A review of agent-based programming for multi-agent systems. Computers **10**(2), 16 (Jan 2021). https://doi.org/10.3390/computers10020016
10. Collier, R.W., Russell, S., Lillis, D.: Reflecting on agent programming with agentspeak(l). In: Chen, Q., Torroni, P., Villata, S., Hsu, J., Omicini, A. (eds.) PRIMA 2015: Principles and Practice of Multi-Agent Systems. pp. 351–366. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-25524-8_22
11. Cranefield, S., Oren, N., Vasconcelos, W.W.: Accountability for practical reasoning agents. In: Lujak, M. (ed.) Agreement Technologies. pp. 33–48. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17294-7_3
12. Dennis, L., Fisher, M., Hepple, A.: Language constructs for multi-agent programming. In: Sadri, F., Satoh, K. (eds.) Computational Logic in Multi-Agent Systems. pp. 137–156. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88833-8_8
13. Dennis, L.A.: Gwendolen semantics: 2017. Tech. Rep. ULCS-17-001, University of Liverpool, Department of Computer Science (2017)
14. Duff, S., Thangarajah, J., Harland, J.: Maintenance goals in intelligent agents. Computational Intelligence **30**(1), 71–114 (2014). https://doi.org/https://doi.org/10.1111/coin.12000, https://onlinelibrary.wiley.com/doi/abs/10.1111/coin.12000

15. Joe-Wong, C., Sen, S., Lan, T., Chiang, M.: Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In: 2012 Proceedings IEEE INFOCOM. pp. 1206–1214. IEEE (2012). https://doi.org/10.1109/INFCOM.2012.6195481
16. Logan, B.: An agent programming manifesto. International Journal of Agent-Oriented Software Engineering **6**(2), 187–210 (2018)
17. de Oliveira Gabriel, V., Panisson, A.R., Bordini, R.H., Adamatti, D.F., Billa, C.Z.: Reasoning in bdi agents using toulmin's argumentation model. Theoretical Computer Science **805**, 76–91 (2020). https://doi.org/https://doi.org/10.1016/j.tcs.2019.10.026, https://www.sciencedirect.com/science/article/pii/S0304397519306553
18. van de Poel, I.: The relation between forward-looking and backward-looking responsibility. In: Vincent, N.A., van de Poel, I., van den Hoven, J. (eds.) Moral Responsibility: Beyond Free Will and Determinism, pp. 37–52. Springer Netherlands, Dordrecht (2011). https://doi.org/10.1007/978-94-007-1878-4_3, https://doi.org/10.1007/978-94-007-1878-4_3
19. Rao, A.S., Georgeff, M.: BDI agents: From theory to practice. In: Proc. 1st Int. Conf. Multi-Agent Systems (ICMAS). pp. 312–319. AAAI, San Francisco, USA (jun 1995)
20. Rao, A.S.: Agentspeak(l): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1038, pp. 42–55. Springer, Berlin, Heidelberg (1996). https://doi.org/10.1007/BFb0031845, https://doi.org/10.1007/BFb0031845
21. Shams, Z., Vos, M.D., Oren, N., Padget, J.: Argumentation-based reasoning about plans, maintenance goals, and norms. ACM Trans. Auton. Adapt. Syst. **14**(3) (Feb 2020). https://doi.org/10.1145/3364220, https://doi.org/10.1145/3364220
22. Vieira, R., Moreira, A., Wooldridge, M., Bordini, R.H.: On the formal semantics of speech-act based communication in an agent-oriented programming language. J. Artif. Int. Res. **29**(1), 221–267 (Jun 2007)
23. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: Proceedings of the Eights International Conference on Principles of Knowledge Representation and Reasoning. p. 470–481. KR'02, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
24. Yazdanpanah, V., Dastani, M., Fatima, S., Jennings, N.R., Yazan, D.M., Zijm, H.: Multiagent task coordination as task allocation plus task responsibility. In: Bassiliades, N., Chalkiadakis, G., de Jonge, D. (eds.) Multi-Agent Systems and Agreement Technologies. pp. 571–588. Springer International Publishing, Cham (2020)
25. Yazdanpanah, V., Dastani, M., Jamroga, W., Alechina, N., Logan, B.: Strategic responsibility under imperfect information. In: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems. p. 592–600. AAMAS '19, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2019)
26. Yazdanpanah, V., Gerding, E.H., Stein, S., Dastani, M., Jonker, C.M., Norman, T.J.: Responsibility research for trustworthy autonomous systems. In: Dignum, F., Lomuscio, A., Endriss, U., Nowé, A. (eds.) AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021. pp. 57–62. ACM, Richland, SC (2021), https://dl.acm.org/doi/10.5555/3463952.3463964