ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230336

Failure Handling in BDI Plans via Runtime Enforcement

Angelo Ferrando[a;*] **and Rafael C. Cardoso**[b]

[a]University of Genova, Genova, Italy
[b]University of Aberdeen, Aberdeen, United Kingdom
ORCiD ID: Angelo Ferrando https://orcid.org/0000-0002-8711-4670, Rafael C. Cardoso https://orcid.org/0000-0001-6666-6954

Abstract. Engineering a software system can be a complex process and prone to failure. This is exacerbated when the system under consideration presents some degree of autonomy, such as in cognitive agents. In this paper, we use runtime verification as a way to enforce safety properties on Belief-Desire-Intention (BDI) agents by enveloping certain plans in safety shields. These shields function as a failure handling mechanism, they can detect and avoid violations in shielded plans. The safety shields also provide automated failure recovery by attempting alternative execution paths to avoid violations.

1 Introduction

The Belief Desire Intention (BDI) model [26] is one of the most popular architectures for programming cognitive agents [11, 12]. This model consists of defining *beliefs*, *goals*, and especially *plans*, which are combined to form the agent's reasoning process. Through such plans, the developer has complete control over the agent. However, the resulting programming process is not trivial. BDI languages, such as AgentSpeak(L) [25], are notoriously different from traditional programming languages and usually come with a steep learning curve. The process of testing [32], debugging [33], and verifying [14] such systems can be quite complex. Solutions which make the BDI development more reliable are of uttermost importance.

BDI agents can execute in dynamic environments, where it may be difficult to guarantee that their behaviour will always be consistent with the developers' expectations. There are various approaches to verify agents' behaviour, but the majority employs static verification. In such scenarios, the agent needs to be abstracted, or some formal representation of it has to be synthesised to be formally verified [14, 23]. This approach is suitable for specific scenarios, where the complexity of states is small and the resulting verification problem is tractable. But, when the agents grow in complexity, more lightweight approaches are required. Runtime Verification (RV) [5, 21] can be a solution to this problem, since monitors can be synthesised starting from formal properties and verified at runtime while the BDI agents are running. No formal representation (nor abstraction) of the system is necessary. RV is focused on detecting unexpected behaviours, rather than enforcing the system to actually behave in a correct way. Enforcing a behaviour leads to Runtime Enforcement (RE) [18]. In this perspective, safety shields combine RV and RE for safe plan execution in BDI agents.

* Corresponding Author. Email: angelo.ferrando@unige.it

We synthesise runtime monitors (called safety shields) to enforce the correct behaviour of existing BDI plans. In this paper, we describe the main features of such safety shields, along with their generation and formal integration into the BDI architecture. We also show how these safety shields can be used to detect wrong behaviour, and how the resulting failure recovery plan can be automatically synthesised in order to find a different plan execution satisfying the safety shield formal property.

A safety shield works as a failure handling mechanism for the plans of the agent. First, it monitors shielded plans by performing failure detection, and then it attempts to perform an automated failure recovery procedure. Every command performed in the agent's shielded plans are checked by their respective safety shields before being executed. In this way, in case the command would violate the safety specification, the safety shield can intervene and stop such command from being completed. Once such a violation is detected, then after stopping the command the shield (if possible) backtracks to a state before the execution of the plan and attempts an alternative plan (if any) for achieving its goal.

A BDI plan is considered to be applicable if its conditions (context) hold at the time that the plan is being considered for execution. While the properties that can be specified using safety shields will oftentimes share similarities with the conditions of the plan, there are two important distinctions. The first is that a shield keeps checking the conditions specified in its formal property during the entirety of the plan's (and any sub-plans) execution; in contrast, the context of a plan checks the condition only before the plan starts executing. The second difference is that the shield specification is more expressive than a context specification, it allows the use of temporal operators from Linear Temporal Logic (LTL) [24] and external actions to formulate a safety property; meanwhile context conditions are restricted to the use of beliefs only.

2 Related Work

For the idea of safety shields, we take inspiration from [1, 17], where safety shields are used to enforce safety properties in reinforcement learning agents. In their work, the shields enforce the formal property by penalising the reward function associated to commands violating the property. No failure handling mechanism is defined whatsoever. This is mainly due to the absence of a symbolic representation of their agents, which is instead accessible for BDI agents in our work.

Our shields are based on the notion of runtime monitors. This is not the first time RV is applied to MAS [2, 19, 28, 22, 30]. However, they usually target agent interaction protocols where the runtime monitors are used to verify the message exchange amongst the agents. This is different to our work, which is verifying internal information of the agent at runtime and enforcing the behaviour to comply with the specification of the shielded plan.

In the standard BDI literature, the most basic approach to automated failure recovery of actions is to repeatedly retry the failed action until it succeeds [34]. A typical example of this is [20], where the failed plan is re-checked to test whether the conditions responsible for the failure no longer hold. Another example can be found in [35], where the authors propose an alternative approach to recover from failures which relies on exploiting interactions between an agent's intentions. Other approaches include basic support for failure handling in Jason [10] that allows developers/users to add their own recovery plans, and a theory for a built-in goal-failure handling mechanism in CANPLAN [29] where an alternative applicable plan is tried when a plan for achieving a goal fails. All of these approaches are simple and effective, but often require additional expert knowledge from the developer. Safety shields are automated, have more expressive conditions, and deal with both failure detection and failure recovery.

Maintenance goals refer to maintaining a constant success state of a goal, in contrast with achievement goals which only require the goal state to be achieved once [15]. Proactive maintenance goal mechanisms, such as in [16], predict that the maintenance condition will be violated in the future and act accordingly in order to avoid the violation. While proactive maintenance goal mechanisms are similar to the functionality of safety shields in terms of handling plan failures, there are some distinct differences: (i) safety shields use formal temporal properties for monitoring the failure condition of plans; and (ii) safety shields do not add any new know-how on what to do in case of failures, it simply excludes options that would violate the shield.

3 AgentSpeak(L)

In this section, we recall the theoretical background which serves as a basis for our novel extensions. AgentSpeak(L) is a logic programming language that provides an abstract framework for programming BDI agents [25]. In this paper, we follow a presentation of AgentSpeak(L) syntax and operational semantics similar to [31], and subsequently in [10] where the failure handling variant was integrated.

The beliefs of an agent determine what an agent currently knows about itself, the other agents in the system, and the environment. They are defined as atomic formulae, as follows:

$$b \ ::= \ P(t_1, \ldots, t_n) \quad (n \geq 0)$$

where P denotes a predicate symbol, and $t_1, \ldots, t_n$ are standard terms of first-order logic. A belief base is a sequence of beliefs:

$$\mathit{beliefs} \ ::= \ b_1 \ldots b_n \quad (n \geq 0)$$

The beliefs defined by the programmer at design time make up for the initial belief base. The rest of the beliefs are then added dynamically during the agent's lifetime.

An achievement goal in AgentSpeak(L) is specified as:

$$g \ ::= \ !at$$

where at is an atomic proposition.

Finally, an action in AgentSpeak(L) is defined as:

$$a \ ::= \ A(t_1, \ldots, t_n) \quad (n \geq 0)$$

Action are written using the same notation as predicates, except that an action symbol A is used instead of a predicate symbol.

Plans are used to define the course of action for the agent to fulfil its goals. A plan has three main components: a triggering event te, denoting the event triggering the execution of the plan, a context $ctxt$, denoting the conditions that must hold to consider the plan applicable, and a body h consisting of a sequence of steps to be executed. A plan in AgentSpeak(L) is defined as:

$$p \ ::= \ te : ctxt \leftarrow h$$

The triggering event is defined as follows:

$$te \ ::= \ +b \mid \ -b \mid \ +g \mid \ -g$$

Specifically, it can be the addition (resp. deletion) of a belief b, and the addition (resp. deletion) of a goal g. A plan is relevant for a triggering event if the event can be unified with the plan's head.

For a plan to be considered applicable a condition $ctxt$ must hold as a logical consequence of the agent's belief.

The body of a plan is composed of actions (a), belief updates ($+b$, $-b$), and achievement goals (g). We omit test goals for brevity, their handling in our approach is the same as for achievement goals. The sequence of formulae denoting the body of a plan is:

$$h \ ::= \ a \mid g \mid \ +b \mid \ -b \mid h; h'$$

An agent program contains a plain library with a set of plans:

$$\mathit{plans} \ ::= \ p_1 \ldots p_n \quad (n \geq 1)$$

Finally, we define an agent through its beliefs and plans:

$$\mathit{agent} \ ::= \ \mathit{beliefs} \ \ \mathit{plans}$$

Definition 1 *Given a set of plans of an agent and a triggering event te, the set $RelPlans(plans, te)$ of relevant plans is:*

$$\{p\phi \mid p \in plans \ \wedge \ \phi = mgu(te, TE(p))\}$$

with $\phi = mgu$ the most general unifier and $TE(p)$ the triggering event of the plan p.

Definition 2 *Given a set of relevant plans R, and the beliefs of an agent, the set of applicable plans $AppPlans(R, beliefs)$ is:*

$$\{p\phi \mid p \in R \ \wedge \ \exists_\phi . beliefs \models ctxt(p)\phi\}$$

with $ctxt(p)$ being the context of plan p.

Relevant plans are all plans that could be triggered by the triggering event te. Applicable plans are the subset of the relevant plans that could be executed considering the agent's current state of mind.

An AgentSpeak(L) configuration C is a tuple $\langle I, E, A, R, Ap, \iota, \rho, \epsilon \rangle$ where: I is the set of intentions $\{i, i', \ldots\}$, with i as an intention stack of partially instantiated plans $[p_1 | p_2 \ldots p_n]$. We use the $|$ symbol to separate plans in an intention stack. E is a set of events $\{\langle te, i \rangle, \langle te', i' \rangle, \ldots\}$. Each event is a pair $\langle te, i \rangle$, where te is a triggering event and i is an intention stack containing plans associated with te. A is a set of actions $\{\langle a, i \rangle, \langle a', i' \rangle, \ldots\}$. Each action is a pair $\langle a, i \rangle$, where a is an action and i is an intention stack containing plans associated with a. R is a set of relevant plans. Ap is a set of applicable plans. ι, ϵ and ρ keep the record of a particular intention, event and applicable plan (respectively) being considered in the current agent's reasoning cycle. Note that to improve readability we have compressed the representation of the inference rules used in the operational semantics in [31, 10], moving some of the elements to C and omitting others that were not used or changed in our extended rules.

To keep the notation compact, we adopt the following: (i) if C is an AgentSpeak(L) configuration, we write C_I to make reference to the component I of C (same for the other components of C, such as C_E and so on); (ii) we write $C_\iota = \_$ to indicate there is no intention considered in the agent's execution (same for C_ρ and C_ϵ); and (iii) we write $i[p]$ to denote the intention stack that has p on its top.

The reasoning cycle of the Jason variant of AgentSpeak(L) has 10 steps, as shown Figure 1. The names are used during the construction of the inference rules to identify at which step the rule can be triggered, and which step it transitions to after applying the rule.

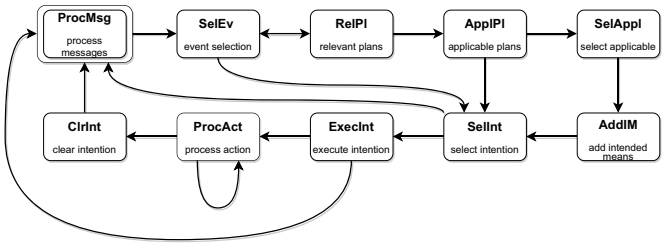

Figure 1. Standard reasoning cycle steps of Jason, introduced in [31] and extended in [10] to include the new step (*ProcAct*). The double rounded rectangle represents the first step in the reasoning cycle.

4 Safety Shields

We extend the AgentSpeak(L) operational semantics to add support for adding/removing safety shields to a plan, and to perform failure detection/recovery with shields. Due to space constraints we only show the rules that were extended, the others can be found in [31, 10].

A shield is a component which can be attached to an agent's plan to check whether such plan violates a formal specification during its execution. In such case, the shield enforces the plan to conform. This allows us to verify aspects concerning each plan's execution, such as the actions it performs, and how it updates the agent's belief base.

Definition 3 *A shield is specified as a tuple $\langle \sigma, \varphi, i, ids \rangle$, where σ is a sequence of events (i.e., actions or addition/removal of beliefs) previously observed during the shielded plan's execution, φ is the formula to be verified (expressed in a formalism of choice), i is the associated intention stack for the shielded plan, and ids is a sequence of identifiers that are automatically generated based on plans that were previously observed during the shielded plan's execution. Given a shield $s = \langle \sigma, \varphi, i, ids \rangle$, we refer to its elements directly as s_σ, s_φ, s_i, and s_{ids}, respectively.*

We specify safety shields by annotating the plans that we want to shield. Annotating plans is a common practice in existing BDI languages [9, 13]. An annotation is a structured label attached to a plan. More formally, a shield annotation can be specified as follows:

$$@shield[\varphi]$$

where $shield$ is a unique label (*e.g.*, $shield_1$, $shield_2$, etc.), and φ is the formal property the shield will enforce. In our implementation we use LTL to specify the properties, but our theory is independent of the formalism used for the property. By design, annotations do not have any specific semantics. The agent's reasoning cycle does not consider them, unless the developer explicitly modifies it to do so.

4.1 Adding and Removing Safety Shields

Shields are attached to the corresponding plan intention stack in C_I. The shield is used only to analyse events concerning corresponding intentions. This can be done by extending the definition of C_I, from a set of intention stacks to a set of tuples $\langle i, S \rangle$, with i an intention stack and S a set of attached shields.

The first set of inference rules of the AgentSpeak(L) semantics that we need to consider concerns the addition of intended means to the set of intentions to be executed, called $ExtEv$ and $IntEv$. In the standard AgentSpeak(L) semantics these rules are as follows:

$$(ExtEv)\frac{}{C, AddIM \rightarrow C', SelInt}\ C_\epsilon{=}\langle te, \top\rangle,\ C_\rho{=}\langle p, \phi\rangle$$

$$where \qquad C'_I = C_I \ \cup\ \{[p\phi]\}$$

$$(IntEv)\frac{}{C, AddIM \rightarrow C', SelInt}\ C_\epsilon{=}\langle te, i\rangle,\ C_\rho{=}\langle p, \phi\rangle$$

$$where \qquad C'_I = C_I \ \cup\ \{i[p\phi]\}$$

In $ExtEv$, external perceptions are handled as events which trigger a new plan p. Since the triggering event is external, no intention (denoted with $\top$ in C_ϵ) needs to be suspended and the plan is added to C_I. In $IntEv$, internal events (*e.g.*, executions of other plans from the same agent) are handled. In such case, the current intention i has to be suspended by adding p to the top of the stack. Both rules originate in the reasoning cycle step $AddIM$, where an instance of a selected plan becomes an "intended means" (*i.e.*, it is added to the set of intentions), and both rules transition to the $SelInt$ step where an intention will be selected for execution.

To keep track of the annotated plans (*i.e.*, shielded plans), we need to override the previous rules as follows (we use the prime symbol $'$ to identify new or extended rules and updated sets):

$$(ExtEv')\frac{Annot(p) = @shield[\varphi]}{C, AddIM \rightarrow C', SelInt}\ C_\epsilon{=}\langle te, \top\rangle,\ C_\rho{=}\langle p, \phi\rangle$$

$$where \quad C'_I = C_I \ \cup\ \{\langle [p\phi], \{\langle [], \varphi, \top, []\rangle\}\rangle\}$$

$$(IntEv')\frac{Annot(p) = @shield[\varphi]}{C, AddIM \rightarrow C', SelInt}\ C_\epsilon{=}\langle te, i\rangle,\ C_\rho{=}\langle p, \phi\rangle,\ \langle i, S\rangle{\in}C_I$$

$$where \quad C'_I = C_I \ \cup\ \{\langle i[p\phi], S \cup \{\langle [], \varphi, i, []\rangle\}\rangle\}$$

Since now plans can have shields, the state of the agent is defined by an extended version of C, where intentions keep track of which shields are currently active. Note that, all other rules in AgentSpeak(L)'s operational semantics are implicitly modified because now C_I consists in a set of tuples, not just intentions. Moreover, due to space constraints we omit the cases when a plan is not annotated, however, the behaviour would be the same as $IntEv$ and $ExtEv$, but with the condition that p is not annotated.

In the extended version $ExtEv'$, we add the shields associated with p directly, since there is no suspended intention attached. In case the plan p is not annotated with a shield, the attached set would be empty, and the resulting tuple in C_I would contain the empty set as second argument in the newly added tuple. Instead, in the extended version $IntEv'$, since an existing intention i is updated by adding p on top of it, we also have to consider the shields that are already present at the intention level. We call S the set denoting the shields already attached to i. To update the set of shields now attached to the updated intention $i[p\phi]$, we add the new shield derived by p to the set of already active shields S. If the new selected plan p is not annotated, then it works as in the previous extended rule $ExtEv'$.

Each shield is attached to an intention stack. An agent can have multiple shields active at the same time. However, each shield is concerned only with the commands executed in its own stack, without being affected by intentions from other stacks and their shields.

The shield is added when the plan is selected to be executed, and it is removed upon plan completion. We can remove the shield by using the rules which are meant to clear completed intentions. When a plan p is considered completed, denoted as an empty intention, the attached shield is removed.

First, we consider the standard rules:

$$(ClearInt_1)\frac{}{C, ClrInt \rightarrow C', ProcMsg}\ C_\iota{=}[hd{\leftarrow}\,]$$

$$where \qquad C'_\iota = \_$$
$$C'_I = C_I \ \setminus\ \{C_\iota\}$$

$$(ClearInt_2)\frac{}{C, ClrInt \rightarrow C', ClrInt}\ C_\iota{=}i'[hd' \leftarrow !at; h' \mid hd \leftarrow]$$

$$\begin{array}{ll} where & C'_\iota = \_ \\ & C'_I = (C_I \setminus \{C_\iota\}) \cup \{i'[hd' \leftarrow h']\} \end{array}$$

$ClearInt_1$ removes an intention from the set of intentions of an agent when there is nothing left in that intention. $ClearInt_2$ removes from the intention what is left from the plan that had been put on the top of the intention stack on behalf of the achievement goal $!at$, which is also removed as it has been accomplished. hd represents the head of the plan, and h' the remaining of the plan.

We can have one (or multiple) shields attached to an intention stack. Since the set of shields is attached to an intention, we do not need to extend $ClearInt_1$, because it handles the case when the intention is completed and completely removed from C_I. Instead, we have to extend $ClearInt_2$, because that is when the intention is not finished, but some of the shields added during the execution might not be necessary anymore.

$$(ClearInt'_2)\frac{RmShields(i'[hd' \leftarrow !at; h'], S) = S''}{C, ClrInt \rightarrow C', ClrInt}$$

$$\begin{array}{ll} where & C_\iota = \langle i'[hd' \leftarrow !at; h' \mid hd \leftarrow], S\rangle \\ & C'_\iota = \_ \\ & C'_I = (C_I \setminus \{C_\iota\}) \cup \{\langle i'[hd' \leftarrow h'], S'\rangle\} \\ & S' = S \setminus S'' \end{array}$$

To define the new version of the rule, we apply an auxiliary function called $RmShields$, which returns the set of removable shields.

Definition 4 *Given an intention stack i of an agent, and a set of shields S, the set of removable shields is given as follows:*

$$RmShields(i, S) = \{\langle s \mid s \in S \wedge s_i = i\}$$

When a shield is added to S, the corresponding suspended intention stack i for the plan is added along with the shield. A shield can be removed when the current intention stack is the same as before the shield was added. This means that the shielded plan concluded its execution without violating the formal specification attached to the shield, and can now be safely removed.

4.2 Catching Violations (Failure Detection)

The entire agent's reasoning cycle depends on which plans are selected as relevant and, then, applicable. We enforce the satisfaction of a formal property by extending the $RelPlans$ function (see Section 3). The goal of such an extension is to take a property into consideration while selecting the relevant plans for a triggering event. The updated function is as follows:

$$\begin{array}{c} RelPlans(plans, te, S) = \\ \{p\phi \mid p \in plans \ \wedge \ \phi = mgu(te, TE(p)) \ \wedge \\ \forall_{s \in S}.(s_\sigma \cdot te \models s_\varphi \ \wedge \ depth(s_i, d) \ \wedge \ id_p \neq s_{ids}(d))\} \end{array}$$

where S denotes the set of shields associated to the current selected intention, and $\cdot$ denotes the concatenation amongst trace of events. Using the updated function, we can check whether the triggering event te violates at least one shield s in S. If that is the case, then $RelPlans$ returns the empty set. Otherwise, if te is conformant to all active shields, then we still need to check whether the plan has already been selected in the past. This is checked considering the position of the plan inside the intention stack, which we refer to as the *depth* of the plan. A plan can be called more than once during the agent's execution, therefore we need the depth of the plan to recognise if the current plan is a recursive call or a duplicate caused by the plan's failure. If that is the case, the plan is not added to the set of relevant plans. This step is necessary to enforce the reasoning cycle to retry the violating plan when a shield is violated. By checking if we have already tried a plan at a specific depth of the intention stack, we can avoid to endlessly loop over the same plan. To obtain this, we need a way to identify the plans. The notation id_p is an abbreviation that we use to uniquely determine a plan. This can be obtained through a direct mapping, that assigns each plan in the plan's library to an identifier.

Note that, when the triggering event (te) violates at least one shield in S, $RelPlans$ returns the empty set. Thus, no relevant or applicable plan is available ($C_R = \emptyset$, $C_{Ap} = \emptyset$). Consequently, no plan can be selected and the resulting plan failure handling is triggered. As shown in $Appl$ rule, this is achieved by adding the corresponding plan deletion event ($-\%at$).

$$(Appl)\frac{AppPlans(C_R, beliefs) = \emptyset}{C, ApplPl \rightarrow C', SelInt}\ C_\epsilon{=}\langle te, i\rangle,\ C_{Ap}{=}\emptyset,\ C_R{\neq}\emptyset$$

$$where\ C'_E = \begin{cases} C_E \cup \{\langle -\%at, i\rangle\} & \text{if } te = +\%at \text{ with } \% \in \{!\} \\ C_E \cup \{C_\epsilon\} & \text{otherwise} \end{cases}$$

By updating $RelPlans$ to consider a formal specification in the plan selection, we can enforce the reasoning cycle to only consider events which do not violate a certain property.

In $SelAppl$, we find the standard rule to select a plan amongst the applicable ones. The S_{Ap} selection function is used to pick one plan amongst the set of applicable ones in C_{Ap} (usually the first).

$$(SelAppl)\frac{S_{AP}(C_{AP}) = \langle p, \phi\rangle}{C, SelAppl \rightarrow C', AddIM}$$

$$where \qquad C'_\rho = \langle p, \phi\rangle$$

We extend this rule in $SelAppl'$, which is necessary to keep track of the events related to active shields. We use this extended rule to update the set of shields. If the triggering event is not a violation of any safety shield, the resulting applicable plan is selected and stored in C_{Ap}, as in the standard semantics. The triggering event is added to the sequence of observed events (used to check against the shield's safety property), and the identifier for the corresponding selected plan is added to the list of the shield's identifiers.

$$(SelAppl')\frac{S_{AP}(C_{AP}) = \langle p, \phi\rangle}{C, SelAppl \rightarrow C', AddIM}$$

$$\begin{array}{ll} where & C'_\rho = \langle p, \phi\rangle \\ & C'_I = (C_I \setminus \{\langle i, S\rangle\}) \cup \{\langle i, S'\rangle\} \\ & S' = \{\langle \sigma', \varphi, i', ids'\rangle \mid \langle \sigma, \varphi, i', ids\rangle \in S \wedge \\ & \qquad \sigma' = \sigma \cdot te \ \wedge \ ids' = ids \cdot id_p\} \end{array}$$

$RelPlans$ only considers triggering events originated from goals and beliefs, therefore, the agent can still perform actions which may violate the formal specification. To enforce actions we need to extend the relevant rule in the operational semantics to handle action violations (*i.e.*, when an action violates a formal property). The $ExecAct$ is the rule that deals with the execution of an action, it was originally extended in [10] as follows:

$$(ExecAct)\frac{\langle a, i\rangle \in C_A \quad execute(a) = e}{C, ProcAct \rightarrow C', ProcAct}$$

$$\begin{array}{ll} where & C'_A = C_A \setminus \{\langle a, i\rangle\} \\ & C'_I = C_I \cup \{i'[te : ct \leftarrow h]\}, \text{if } e \\ & C'_E = C_E \cup \{\langle -\%at, i\rangle\}, \text{if } \neg e \wedge (te = +\%at) \\ with & i = i'[te : ct \leftarrow a; h] \text{ and } \% \in \{!\} \end{array}$$

This rule describes how the agent's reasoning cycle proceeds when an action is executed. It uses the auxiliary Boolean function $execute$ to refer to the actual execution of actions. If the action succeeded (e is true), then the attached suspended intention is resumed (if e condition). Otherwise (e is false), the deletion event for the plan calling the action is added to set of the events (if $\neg e$ condition). In the standard operational semantics we also have $ExecDone$, which handles when the action is completed. We omit that rule here since it does not need to be extended.

We update the existing rule to consider the case when the action satisfies the property, and add an additional rule for when the action violates it. In this way, we can enforce the agent to not perform actions that would violate the expected behaviour specified in the property. First, we modify $ExecAct$, and rename it $ExecAct'_1$:

$$(ExecAct'_1)\frac{\langle a, i\rangle \in C_A \quad execute(a) = e}{C, ProcAct \rightarrow C', ProcAct}\ {}_{\langle i,S\rangle \in C_I,\ \forall_{s\in S}.s_\sigma \cdot a \models s_\varphi}$$

$$\begin{array}{ll} where & C'_A = C_A \setminus \{\langle a, i\rangle\} \\ & C'_I = C_I \cup \{\langle i'[te : ct \leftarrow h], S'\rangle\}, \text{if } e \\ & C'_E = C_E \cup \{\langle -\%at, i\rangle\}, \text{if } \neg e \wedge te = +\%at \\ & S' = \{\langle \sigma', \varphi, i', ids\rangle \mid \langle \sigma, \varphi, i', ids\rangle \in S \wedge \sigma' = \sigma \cdot a\} \\ with & i = i'[te : ct \leftarrow a; h] \text{ and } \% \in \{!\} \end{array}$$

$ExecAct'_1$ is used when action a does not violate the specification.

$$(ExecAct'_2)\frac{\langle a, i\rangle \in C_A}{C, ProcAct \rightarrow C', ProcAct}\ {}_{\langle i,S\rangle \in C_I,\ \exists_{s\in S}.s_\sigma \cdot a \not\models s_\varphi}$$

$$\begin{array}{ll} where & C'_A = C_A \setminus \{\langle a, i\rangle\} \\ & C'_E = C_E \cup \{\langle -\%at, i\rangle\}, \text{if } te = +\%at \\ with & i = i'[te : ct \leftarrow a; h] \text{ and } \% \in \{!\} \end{array}$$

In $ExecAct_2$, the violation of the specification s_φ (for some $s \in S$) is used to trigger the deletion event of the plan calling the violating action. This aspect is relevant from an engineering perspective, since we can reuse the failure handling mechanism from [10].

4.3 Handling Violations (Failure Recovery)

Recovery plans that are triggered in the plan failure mechanism are domain dependent. It is up to the developer to define such behaviour and how the agent should proceed in its plan selection. Usually, this consists in executing some cleaning actions/plans (to clear off the intermediate modifications made by the plan before failing), and then trying to achieve the failed plan's goal through a different plan.

To perform automated failure recovery we need to identify what are the requirements to automate such synthesis. To try to automate the generation of recovery plans for general use we split it into two parts: *restore* and *retry*. Before we discuss both parts in more detail, we add an additional inference rule to the operational semantics:

$$(IntEvFail')\frac{C_\epsilon = \langle -!at, i\rangle}{C, AddIM \rightarrow C', SelInt}\ {}_{\langle i,S\rangle \in C_I,\ \exists_{s\in S}.s_\sigma \not\models s_\varphi}$$

$$where \quad C'_I = C_I \cup \{i[!at][cmds] \mid restore(s_\sigma) = cmds\}$$

In this rule, we automatically handle the case of failure caused by shield violation. It is applied when the triggering event is a plan deletion, and at least one shield attached to the intention has been violated. If that is the case, then we need to *restore* the agent's mind, and *retry* to achieve the goal by attempting to follow a different execution path (recalling $!at$ in this case). $cmds$ contains the commands to restore.

Restore. When an agent executes a shielded plan, some commands from the plan's body may change the agent's state of mind or the environment. Thus, when a shielded plan is about to violate the shield's formal property (φ), the restoring process consists in reverting the beliefs and actions which have been done since the shielded plan started.[1] Note that, we restore only what was done inside the shielded plan; beliefs or actions could have been changed due to other concurrent plans not related to the shield or other agents in the system which are outside the scope of this mechanism. More complex situations may require domain specific restore mechanisms.

Even though the act of restoring an action might be very intuitive in some scenarios, it might not be as much in others (*e.g.*, where an action can be restored only by a combination of different actions or cannot be restored at all). The act of restoring an action is domain specific, and it needs to be customised by the developer. We use a semi-automatic approach by using a list of opposite actions provided by the developer. We also assume that we can extract the effects of an action either through the action's specification (uncommon in BDI languages) or through the environment. In plans without actions our process remains completely automated.

The restore function can be defined as follows:

$$restore(\sigma) = [opposite(\sigma(|\sigma|)), \ldots, opposite(\sigma(1))]$$

where, given a list of events, it returns a list of opposite commands (in backward order). This list is used in the $(IntEvFail')$ rule to restore the agent's mind.

Retry. Once the plan has been restored, the agent can retry to achieve the same goal. In rule $(IntEvFail')$, after restoring the agent's mind we call the same plan again. In the standard operational semantics this could result in an endless loop. However, as previously anticipated, we modified $RelPlans$ to keep track of which plans have already been tried. If no more plans are available, or all available ones have already been tried, then the plan would simply fail and fall outside the scope of the shield. In such cases, the default behaviour takes over and a domain-specific failure plan will be selected if available.

5 Implementation

We implemented a prototype in the JaCaMo multi-agent development framework [8, 7].[2] From an implementation perspective, extending the reasoning cycle of Jason is an invasive task, which requires altering its source code. Instead, we focus on a less invasive and more portable implementation based on instrumentation. Jason supports annotations and governs the agent programming dimension as part of JaCaMo. The environment in JaCaMo is defined through CArtAgO [27], which uses artefacts to describe *observable properties* (*i.e.*, perceptions) about particular elements of the environment as well as to provide *operations* (*i.e.*, actions) that the agent can execute. In this work we do not make use of the third dimension in Ja-

[1] Some actions may not be reverted. The only side effect in such cases is that it may lead to less (or zero) possible execution paths, therefore reducing the probability of success of the automated retry mechanism.

[2] Source code available at https://github.com/AngeloFerrando/SafetyShieldsBDI (Accessed 21-July-2023)

CaMo, which is related to organisations. There are other approaches in the literature that can deal with robustness at the organisation level in multi-agent systems [4] and more specifically in JaCaMo [3].

We use JaCaMo instead of Jason because the former supports artefacts which are well-suited for implementing the shields and interfacing with the monitors. We create one CArtAgO artefact per agent which is exclusively responsible for maintaining all of the information about the shields, and any shield-related operation such as when a shield needs to be added, removed, or updated. The monitors are implemented in LamaConv[3], a Java library for LTL monitors.

5.1 Adding and Removing Safety Shields through Instrumentation

This can be obtained by instrumenting the plan by adding an operation to add a shield as the first command in the plan's body, and by adding another operation to remove the same shield as the last command in the plan. Thus, if we have a generic plan as follows:

```
@shield1[φ]
+!plan : ctxt <- cmd1;...;cmdm.
```

after the instrumentation process, we obtain:

```
+!plan : ctxt <-
 add_shield(φ); cmd1;...;cmdm; remove_shield(φ).
```

where we call specific operations to add/remove the shield. The `add_shield` artefact operation is placed at the beginning of the plan's body. This operation adds a shield to a map data structure kept in the agent's artefact. As soon as the plan is selected to be executed, the corresponding shield is added into the agent's artefact. When the plan is considered completed, the previously added shield can be removed with the `remove_shield` operation.

Note that, `remove_shield` is also called when the plan fails. This happens when the retry phase does not successfully recovers the plan. This is necessary to ensure that the shield is always removed, both when the plan succeeds, as well as when it fails.

5.2 Catching Violations through Instrumentation

This can be obtained by explicitly checking, for each command in the shielded plan's body, whether the command would violate any safety property in S. The instrumentation process adds the artefact operation `update_shields` before each command. Considering the previous plan, the final result after instrumentation is as follows:

```
+!plan : ctxt <-
  add_shield(φ);
  update_shields(cmd1); cmd1;
  ...;
  update_shields(cmdm); cmdm;
  remove_shield(φ).
```

If the command is consistent with all shields associated to the plan, then the operation succeeds. Otherwise, the operation fails and causes the plan to fail as well.

5.3 Restore and Retry through Instrumentation

By using the Jason's internal action (`.intention`), it is possible to store and retrieve information about a specific plan execution. For each plan's execution, a unique ID is stored, along with the depth in which such plan has been called w.r.t. the intention's stack. With these, a plan can be instrumented to remember whether previous executions of a plan have already been called because of a shield failure, which is used to avoid executing it again.

Considering the same plan as before, we would obtain:

```
+!plan : ctxt &
  .intention(I) & id(I, PlanID) &
  depth(I, Depth) &  already_tried(I, Plans) &
  not(member((PlanID, Depth), Plans))
<-
  -already_tried(I, Plans);
  +already_tried(I, [(PlanID, Depth)|Plans]);
  ...
-!plan : .intention(I) & violated(I, Cmds) <-
  !restore(Cmds); !plan.
```

The first part deals with how the instrumentation process adds additional requirements in the plan's context. Other than the original one ($ctxt$), the plan's intention is retrieved (I), as well as the corresponding plan's id ($PlanID$) and depth ($Depth$). These are used to check whether the plan has already been called. The second part consists in restoring and retrying in case of failure (*i.e.,* when a violation is detected). In such recovery plan (`-!plan`), the effects are reverted ($Cmds$ contains the list of commands to be executed to restore the agent's mind) and the plan is retried (`!plan`).

Without the instrumentation, every time a plan is retried it would go through the same execution path. We can avoid this by storing information about previous executions of a shielded plan, ignoring plans already attempted and trying different executions paths (if any).

6 Experiments

We report two categories of experiments: (i) *safety*, where we apply our implementation to a case study; and (ii) *performance*, where we focus on calculating the overhead of using shields.

6.1 Safety Experiments

With a single shield An agent controlling a rover has been deployed in a nuclear facility to perform remote inspection. The rover's mission is to inspect barrels containing nuclear waste, and to report in case of leakage or other structural issues. The agent has full control over the rover's sensors/actuators and its movement. The BDI agent program used in this demonstrative example is as follows:

```
// initial beliefs
fast.
position(wp1).
//plans
@shield1("G(+rad(low))")
+!inspect_nuclear_plant : fast <-
  !inspect(wp1);
  move_to(wp2, R2); -rad(R2); +rad(R2);
  move_to(wp3, R3); -rad(R3); +rad(R3);
  !inspect(wp3).
+!inspect_nuclear_plant : true <-
  !inspect(wp1);
  move_to(wp4, R4); -rad(R4); +rad(R4);
  move_to(wp3, R3); -rad(R3); +rad(R3);
  !inspect(wp3).
+!inspect(WP) : true <-
  inspect_barrel(WP, Result).
```

The initial beliefs correspond to what the agent knows to be true; the agent knows the rover is at `wp1` (`position(wp1)`) and that a `fast` inspection is desired (the shortest path between `wp1` and `wp3`, is through `wp2`). The main plan `!inspect_nuclear_plant` offers two choices. The first one, when the inspection is expected to be

[3] https://www.isp.uni-luebeck.de/lamaconv (Accessed: 21-July-2023)

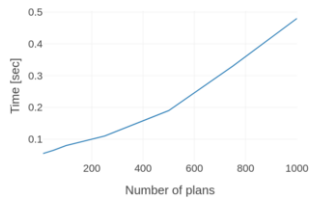

Figure 2. Instrumentation.

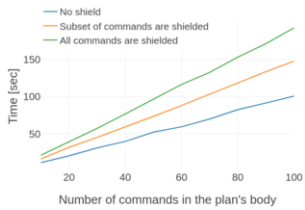

Figure 3. One single plan.

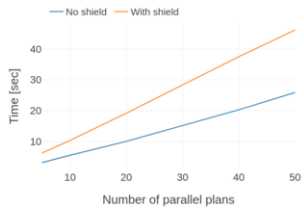

Figure 4. Parallel plans.

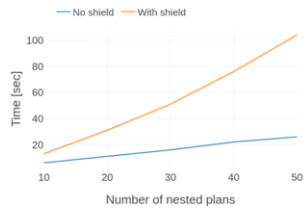

Figure 5. Nested plans.

quick; and the second one, when there is no time constraint. Then, depending on which plan is selected, sub-plans and actions are called.

The sub-plan `!inspect(WP)` makes the rover inspect a certain waypoint `WP`. First, the rover has to move to the waypoint `WP` (`move_to`), then inspect the barrel (`inspect_barrel`). The rover is equipped with a radiation sensor, and when it moves to a waypoint, it detects the level of radiation in that area (`move_to` second parameter). Note that, `!inspect_nuclear_plant` has been annotated with a formal property (LTL), where `G` stands for globally ($\Box$ operator). Such formal property is a safety property stating that: "It is always true that the perceived radiation level is low". This formal property practically says that the rover should not keep moving to the next waypoint if it perceives a high level of radiation, in which case the shield detects a failure. If, during the execution of the first plan, the safety shield perceives high radiation while the agent is attempting to reach a waypoint, it reports a violation of the safety property. This will trigger the restore and retry behaviour. If the high level of radiation was perceived in `wp2`, then the only command to restore is `move_to`; which is done by moving back to `wp1`. After restoring, the retry attempts the second case of `!inspect_nuclear_plant` to achieve the goal of inspecting both waypoints. In this case, by moving through a different path.

The remote inspection example can be achieved without the need of shields. However, the resulting plans would be much more verbose and complex. For instance, if we wanted to obtain the same result without using safety shields, we would have to add an additional check before each single action or addition/removal of a belief. Moreover, this should be done manually for each plan. Instead, with safety shields the additional checks are automatically synthesised, and everything is parametric w.r.t. a formal specification of interest. Thus, if we want to check different requirements for a plan, it is enough to change the formula in the shield; while, without shields, the developer would have to re-write everything from scratch.

With two nested shields Considering the same example as before. We annotate the `!inspect` plan as follows (remember that `!inspect` is a sub-plan of `!inspect_nuclear_plant`):

```
@shield2["(+rad(low) U inspect_barrel(_,ok)) or
     (+rad(high) U inspect_barrel(_,leakage))"]
+!inspect(WP) : true <- ...
```

The property states that: "If the sensor perceives `high` radiation, then the inspection of the barrel in the current waypoint has to report `leakage` (and *viceversa*)". We use this property to ensure that the radiation and visual sensors are consistent. Note that, this LTL formula is more complex, since safety shields support LTL specifications, the properties that can be specified are as expressive as LTL. The presence of a shield in the nested plan is handled by adding an additional shield upon calling `!inspect`. Consequently, when inside the `!inspect` sub-plan, both `shield1` and `shield2` are enforced.

6.2 *Performance Experiments*

Our experiments focus on the overhead that could be introduced by the failure detection mechanism. We use stub actions and sub-plans to simulate a more realistic behaviour. The properties analysed on such plans follow the structure used in Section 6.1.

Figure 2 reports the execution time for instrumenting plans. As expected, the time is linear w.r.t. the number of plans to instrument. Moreover, even with a large number of plans to instrument ($\sim$1000), the required time to complete the process is less than 0.5 seconds, making the instrumentation step usable in realistic scenarios.

In Figure 3, we report the experiments with a single plan. The aim is to show the overhead of adding a shield to a plan with varying size of the plan's body. Each command in the plan's body needs to be reported to the shield. We have the case without a shield, with a subset of shielded commands (half of them), and with all commands shielded. In the last two, we observe the overhead, which depends on the number of commands reported to the shield. Note that, even though the overhead is not negligible, the time complexity remains linear w.r.t. the number of commands. This is derived by the fact that a monitor takes constant time to analyse a single command [6].

The experiments with parallel plans are shown in Figure 4. The number of commands to be executed per plan is fixed (50 commands), varying the number of plans. Similarly to the previous experiment, the overhead is not negligible, but preserves the time complexity. Note that, all plans are shielded (worst case); in a more realistic scenario, only a subset of the plans would be shielded.

In Figure 5, we report the experiments with nested plans, where each plan calls a single sub-plan, that in turn calls another sub-plan, and so on. This scenario is, as expected, the one with the worst performance for shields, since each nested shield increases the number of shields to update by one. This is a very stressed and unlikely scenario, where each nested plan is shielded.

7 Conclusions and Future Work

In this paper, we presented the design and implementation of safety shields for BDI plans. We formally specify how to enhance the agent's reasoning cycle to enforce the satisfaction of safety properties through shields. A prototype of our approach is proposed using the JaCaMo platform. We report experiments using an example of a rover deployed in a safety-critical scenario to show how the safety behaviour is enforced. Moreover, we also reported empirical experiments to show the performance of the resulting prototype.

For future work, we are interested in improving our prototype. The current implementation is based on instrumentation. However, instrumentation has implications at the engineering level, and it is less ideal in the long run w.r.t. the actual agent's reasoning cycle modification. We are also interested in extending the work from using one single artefact per agent, to one artefact per shield. This extension could improve performance, especially with nested shields.

Acknowledgements

This project CONVINCE has received funding from the European Union's Horizon research and innovation programme G.A. n. 101070227. This publication is funded by the European Union. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or European Commission (the granting authority). Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu, 'Safe reinforcement learning via shielding', in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pp. 2669–2678, United States, (2018). AAAI Press.

[2] Davide Ancona, Angelo Ferrando, and Viviana Mascardi, 'Parametric runtime verification of multiagent systems', in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pp. 1457–1459. ACM, (2017).

[3] Matteo Baldoni, Cristina Baroglio, Olivier Boissier, Roberto Micalizio, and Stefano Tedeschi, 'Distributing responsibilities for exception handling in JaCaMo', in *20th International Conference on Autonomous Agents and Multiagent Systems*, pp. 1752–1754. ACM, (2021).

[4] Matteo Baldoni, Cristina Baroglio, Roberto Micalizio, and Stefano Tedeschi, 'Accountability in multi-agent organizations: from conceptual design to agent programming', *Auton. Agents Multi Agent Syst.*, **37**(1), 7, (2023).

[5] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger, 'Introduction to runtime verification', in *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, 1–33, Springer, Cham, (2018).

[6] Andreas Bauer, Martin Leucker, and Christian Schallhart, 'Runtime verification for LTL and TLTL', *ACM Trans. Softw. Eng. Methodol.*, **20**(4), 14:1–14:64, (2011).

[7] O. Boissier, R.H. Bordini, J. Hubner, and A. Ricci, *Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo*, Intelligent Robotics and Autonomous Agents series, MIT Press, United States, 2020.

[8] Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci, and Andrea Santi, 'Multi-agent oriented programming with JaCaMo', *Science of Computer Programming*, **78**(6), 747–761, (Jun 2013).

[9] Rafael Bordini, Jomi Hübner, and Michael Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*, volume 8, John Wiley & Sons, Ltd, United Kingdom, 10 2007.

[10] Rafael H. Bordini and Jomi Fred Hübner, 'Semantics for the Jason variant of AgentSpeak (plan failure and some internal actions)', in *19th European Conference on Artificial Intelligence*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pp. 635–640. IOS Press, (2010).

[11] Rafael H. Bordini, Amal El Fallah Seghrouchni, Koen V. Hindriks, Brian Logan, and Alessandro Ricci, 'Agent programming in the cognitive era', *Auton. Agents Multi Agent Syst.*, **34**(2), 37, (2020).

[12] Rafael C. Cardoso and Angelo Ferrando, 'A review of agent-based programming for multi-agent systems', *Computers*, **10**(2), 16, (Jan 2021).

[13] Stephen Cranefield, Michael Winikoff, Virginia Dignum, and Frank Dignum, 'No pizza for you: Value-based plan selection in BDI agents', in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pp. 178–184. IJCAI, (2017).

[14] Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini, 'Model checking agent programming languages', *Autom. Softw. Eng.*, **19**(1), 5–63, (2012).

[15] Simon Duff, James Harland, and John Thangarajah, 'On proactivity and maintenance goals', in *5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 1033–1040. ACM, (2006).

[16] Simon Duff, John Thangarajah, and James Harland, 'Maintenance goals in intelligent agents', *Comput. Intell.*, **30**(1), 71–114, (2014).

[17] Ingy Elsayed-Aly, Suda Bharadwaj, Christopher Amato, Rüdiger Ehlers, Ufuk Topcu, and Lu Feng, 'Safe multi-agent reinforcement learning via shielding', in *20th International Conference on Autonomous Agents and Multiagent Systems*, pp. 483–491. ACM, (2021).

[18] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier, 'Runtime enforcement monitors: composition, synthesis, and enforcement abilities', *Formal Methods Syst. Des.*, **38**(3), 223–262, (2011).

[19] Angelo Ferrando, Davide Ancona, and Viviana Mascardi, 'Decentralizing MAS monitoring with DecAMon', in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pp. 239–248. ACM, (2017).

[20] Malik Ghallab, Dana S. Nau, and Paolo Traverso, *Automated Planning and Acting*, Cambridge University Press, Cambridge, UK, 2016.

[21] Martin Leucker and Christian Schallhart, 'A brief account of runtime verification', *J. Log. Algebraic Methods Program.*, **78**(5), 293–303, (2009).

[22] Yoo Jin Lim, Gwangui Hong, Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae, 'A runtime verification framework for dynamically adaptive multi-agent systems', in *2016 International Conference on Big Data and Smart Computing, BigComp 2016, Hong Kong, China, January 18-20, 2016*, pp. 509–512. IEEE Computer Society, (2016).

[23] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi, 'MCMAS: an open-source model checker for the verification of multi-agent systems', *Int. J. Softw. Tools Technol. Transf.*, **19**(1), 9–30, (2017).

[24] Amir Pnueli, 'The temporal logic of programs', in *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 46–57. IEEE Computer Society, (1977).

[25] Anand S. Rao, 'AgentSpeak(L): BDI agents speak out in a logical computable language', in *7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996*, volume 1038 of *Lecture Notes in Computer Science*, pp. 42–55. Springer, (1996).

[26] Anand S. Rao and Michael P. Georgeff, 'BDI agents: From theory to practice', in *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pp. 312–319. The MIT Press, (1995).

[27] Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini, 'Environment programming in CArtAgO', in *Multi-Agent Programming: Languages, Tools and Applications*, Multiagent Systems, Artificial Societies, and Simulated Organizations, chapter 8, 259–288, Springer, Boston, MA, (2009).

[28] Chittra Roungroongsom and Denduang Pradubsuwun, 'Formal verification of multi-agent system based on JADE: A semi-runtime approach', in *Recent Advances in Information and Communication Technology 2015*, 297–306, Springer, (2015).

[29] Sebastian Sardiña and Lin Padgham, 'A BDI agent programming language with failure handling, declarative goals, and planning', *Auton. Agents Multi Agent Syst.*, **23**(1), 18–70, (2011).

[30] Paolo Torroni, Pinar Yolum, Munindar P. Singh, Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, and Paola Mello, 'Modelling interactions via commitments and expectations', in *Handbook of Research on Multi-Agent Systems - Semantics and Dynamics of Organizational Models*, 263–284, IGI Global, (2009).

[31] Renata Vieira, Álvaro F. Moreira, Michael J. Wooldridge, and Rafael H. Bordini, 'On the formal semantics of speech-act based communication in an agent-oriented programming language', *J. Artif. Intell. Res.*, **29**, 221–267, (2007).

[32] Michael Winikoff, 'BDI agent testability revisited', *Auton. Agents Multi Agent Syst.*, **31**(5), 1094–1132, (2017).

[33] Michael Winikoff, 'Debugging agent programs with why?: Questions', in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, pp. 251–259. ACM, (2017).

[34] Wayne Wobcke, 'An operational semantics for a PRS-like agent architecture', in *14th Australian Joint Conference on Artificial Intelligence, Adelaide, Australia, December 10-14, 2001*, volume 2256 of *Lecture Notes in Computer Science*, pp. 569–580. Springer, (2001).

[35] Yuan Yao, Brian Logan, and John Thangarajah, 'Robust execution of BDI agent programs by exploiting synergies between intentions', in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pp. 2558–2565. AAAI Press, (2016).