ECAI 2025
I. Lynce et al. (Eds.)

© 2025 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA251228

Reliable Intention Selection in BDI Agents with Recovery Shields

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

[a]University of Modena and Reggio Emilia, Modena, Italy
[b]University of Aberdeen, Aberdeen, United Kingdom
ORCID (Angelo Ferrando): https://orcid.org/0000-0002-8711-4670, ORCID (Rafael C. Cardoso): https://orcid.org/0000-0001-6666-6954

Abstract. The existing approaches to enforcing runtime properties and handling failures in autonomous agents primarily focus on single-agent systems, responding to violations by immediately rejecting unsafe actions. In this paper, we extend the notion of safety shields for Belief-Desire-Intention agents by proposing a revised version where a shield can observe and respond not only to what the agent itself does, but also to changes from other agents, when these affect the shielded agent's behaviour. Our shields suspend intentions that would break a formal specification and resume them once it is safe to do so. To support this, we introduce recovery shields, a new mechanism that defines when a suspended intention can be safely resumed. Our main contributions are extensions to the reasoning cycle and operational semantics of AgentSpeak(L), as well as an implementation in the JaCaMo platform.

1 Introduction

The Belief-Desire-Intention (BDI) model has long served as a foundation for programming cognitive agents [8, 9], offering a high-level abstraction where behaviour is specified through symbolic components such as beliefs, goals, and plans [22]. BDI agent programming languages such as AgentSpeak(L) [21] offer fine-grained control over agent reasoning, but this flexibility comes at the cost of increased complexity in development [11], debugging [27], and especially runtime correctness assurance [12]. As agents operate in increasingly open, dynamic, and concurrent environments, ensuring reliable execution becomes more challenging and critical.

Traditional approaches to validating agent behaviour, such as static verification and model checking [19], typically rely on formal abstractions of agent logic. While effective in constrained settings, such techniques scale poorly when dealing with multiple agents and dynamic environments. In contrast, Runtime Verification (RV) [3] allows formal properties to be monitored during execution, offering a lightweight and adaptable alternative. However, RV is inherently passive, it detects violations but does not intervene to prevent them.

Runtime Enforcement (RE) [15] addresses this limitation by combining monitoring with intervention. Through RE, systems can be constrained to adhere to desired properties at runtime, enabling preventative control rather than post hoc diagnosis. Within this paradigm, *safety shields* [16] are a promising technique for enforcing temporal properties in symbolic agent systems. A shield monitors plan execution, and when a violation is imminent or detected, it blocks unsafe commands, allowing the agent to remain within safe operational bounds.

In this paper, we extend the notion of safety shields in two key ways. First, we enhance the agent's reasoning cycle by allowing intentions to be *suspended*, rather than failed, when a shield detects a violation. These suspended intentions can be *resumed* once conditions become safe again, enabling agents to handle temporary violations without abandoning their goals or having to replan. Second, we introduce *recovery shields*, a new mechanism that complements safety shields by defining the precise conditions under which a suspended intention may safely resume. While safety shields are responsible for enforcement, recovery shields handle resumption. These recovery conditions are automatically synthesised from the violation trace and expressed as temporal properties monitored at runtime.

Our proposal is formalised as an extension to the operational semantics of AgentSpeak(L), with modifications to the intention selection, execution, and recovery rules. We implement our approach within the JaCaMo multi-agent platform [5], which integrates AgentSpeak(L) agents and environment artifacts. The proposed extensions are tested in a personal assistant system involving asynchronous agent interactions and external interference.

2 Preliminaries

In this section, we cover the theoretical background on AgentSpeak(L) and satefy shields required to present our contributions.

2.1 AgentSpeak(L)

AgentSpeak(L) is an abstract language for programming BDI agents [21], which was later implemented and further extended in languages such as Jason [6]. Jason is the programming language used to program agents in the JaCaMo platform. The AgentSpeak(L) syntax and operational semantics we present in this section follow the notation discussed in [26], and subsequently in [7] where it was extended to include failure handling.

An agent program consists of beliefs, goals, and plans. The reasoning cycle processes events by selecting relevant plans, instantiating them with substitutions, and executing them through intentions represented as stacks of partially executed plans. More formally, an

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

agent program is defined through its belief base and plan library:

$$agent \ ::= \ beliefs \ plans$$

The belief base is a sequence of beliefs:

$$beliefs \ ::= \ b_1 \dots b_n \quad (n \geq 0)$$

Note that $(n \geq 0)$ means it can be empty, and $(n \geq 1)$ means there must be at least one.

The initial belief base is generated at the program's start and is usually created by the developer at design time. The remaining beliefs are then added dynamically during the agent's execution. Beliefs are defined as atomic formulae, and represent what an agent currently knows about itself, other agents in the system, and the environment:

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

where P denotes a predicate symbol, and $t_1, \dots, t_n$ are standard terms of propositional logic.

The plan library contains a set of plans:

$$plans \ ::= \ p_1 \dots p_n \quad (n \geq 1)$$

A plan in AgentSpeak(L) is defined as:

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

Plans define the course of action for the agent to fulfil its goals. A plan has a triggering event te, denoting the event that can trigger the plan, a context condition $ctxt$, indicating the preconditions that must hold to consider the plan applicable, and a body h consisting of a sequence of steps to be executed.

The triggering event is defined as follows:

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

Notably, a triggering event can be the addition/deletion of a belief b or a goal g. A plan is relevant for a triggering event if the event can be unified with the plan's head. Relevant plans are all plans that could be triggered by the triggering event te.

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, $p\phi$ plan p under substitution ϕ, and $TE(p)$ the triggering event of the plan p.

For a plan to be considered applicable, a condition $ctxt$ must hold as a logical consequence of the agent's belief base. Applicable plans are the subset of the relevant plans that could be executed considering the agent's current state of mind.

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.

The sequence of formulae denoting the body of a plan is:

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

where body of a plan is composed of actions (a), belief updates ($+b$, $-b$), and achievement goals (g). We omit test goals for brevity, as we handle them in the same way as achievement goals in our approach.

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:

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

where A is a predicate symbol.

An agent configuration C is a tuple $\langle I, E, R, Ap, \epsilon, \rho \rangle$ where: I is the set of intentions $\{i, i', \dots\}$, with i an intention stack of partially instantiated plans $[p_1|p_2 \dots p_n]$. We use the $|$ symbol to separate plans in an intention stack, where the leftmost plan is the active one. E is a set of events $\{\langle te, i \rangle, \langle te', i' \rangle, \dots\}$. 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. R is a set of relevant plans. Ap is a set of applicable plans. ϵ and ρ are the event and applicable plan under consideration in the current agent's reasoning cycle.

To improve readability and keep the notation compact, we have compressed the representation of the inference rules used in the operational semantics from [26, 7], omitting elements that were not used or changed and moving some of the elements to C as follows: (i) 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); and (ii) we write $i[p]$ to denote the intention stack that has p on its top.

2.2 Safety Shields

A shield is a component attached to a plan to check compliance with a formal specification during execution. If a violation is detected, the shield enforces conformance. This enables verification of plan behaviour, including sub-plan calls and belief updates.

Definition 3. *A shield is a tuple $\langle \sigma, \varphi, i \rangle$, where σ is the sequence of observed events (additions/removals of beliefs or goals), φ is the property to verify, and i is the intention stack of the shielded plan. For a shield $s = \langle \sigma, \varphi, i \rangle$, we refer to its elements as s_σ, s_φ, and s_i.*

Safety shields are defined by annotating plans, which is a feature in Jason [6, 10]. Formally, a shield annotation is:

$$@shield[\varphi]$$

where $shield$ is a unique label and φ is the LTL [20] property to be enforced. Annotations are semantically inert unless explicitly interpreted by the reasoning cycle.

Shields are attached to intention stacks in C_I, and apply only to events from their associated intentions. To support this, in [16] C_I was extended to a set of tuples $\langle i, S \rangle$, where i is an intention stack and S its attached shields. Safety shields can be added, removed, and handled by extending the AgentSpeak(L) operational semantics. Due to space constraints, we only present the rules necessary to understand our work. The complete set of safety shield rules can be found in [16].

The $ExtEv$ and $IntEv$ rules add intended means to the intention set. To keep track of the annotated plans (*i.e.*, shielded plans), the following rules are applied:

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

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

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

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

In $ExtEv$, since no intention is suspended, the shield from p (if any) is directly attached; otherwise, the shield set is empty. In $IntEv$, p is pushed onto an existing intention i with shield set S, and any new shield from p is added to S. If p is not annotated, the behaviour matches the original rules from [26].

A shield is attached to a specific intention stack. While multiple shields may be active concurrently, each shield monitors only the commands within its own stack, independent of other intentions and their shields. The shield is added when the plan is selected to be executed, and it is removed upon plan completion.

The agent's reasoning cycle relies on selecting relevant and applicable plans. To enforce formal properties, the $RelPlans$ function (see Section 2.1) is extended to account for property satisfaction during plan selection. The updated function is as follows:

$$RelPlans(plans, te, S) = \{p\phi \mid p \in plans \ \wedge \ \phi = mgu(te, TE(p)) \ \wedge \ \forall_{s \in S}.(s_\sigma \cdot te \models s_\varphi\}$$

Here, S denotes the set of shields associated with the current intention, and $\cdot$ indicates event trace concatenation. The updated function checks whether the triggering event te violates any shield $s \in S$. If so, it returns the empty set. We use the symbol $\models$ to denote LTL satisfaction over event traces in this context. This differs from the earlier use of $\models$ for belief entailment in context conditions; while the notation is the same, the underlying semantics are distinct.

If the triggering event te violates any shield in S, $RelPlans$ returns $\emptyset$, making both C_R and C_{Ap} empty. As a result, no plan can be selected, and failure handling is triggered by adding the corresponding plan deletion event ($-\%at$), as specified by the $Appl$ rule.

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

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

By extending $RelPlans$ to account for formal specifications, the reasoning cycle considers only events that respect the given property. In $SelAppl$, the S_{Ap} function selects a plan from the applicable set C_{Ap}, typically the first one.

In the safety shield context, this rule tracks events relevant to active shields by updating their traces. If te does not violate any shield, the selected applicable plan is stored in C_{Ap}, and te is appended to the shield's event trace along with the selected plan's identifier.

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

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

3 Reliable Intention Selection

In contrast to past work that enforces safety over actions [16], we focus on beliefs and goals as the primary targets of shield monitoring and enforcement. While actions are natural enforcement points, their effects on the environment are often irreversible, unpredictable, and heavily domain-specific. General recovery from unsafe actions, such as physically moving an object or sending a message, is inherently difficult to generalise. Although prior work has explored action-level shields, recovery in such settings requires bespoke logic tied to specific domains. Instead, by concentrating on the internal dynamics of belief updates and goal adoption, our shields can detect violations and recover from failure in a lightweight and automated manner. This enables a reliable intention suspension mechanism that integrates seamlessly with the agent's reasoning cycle to ensure reliable intention selection.

In Figure 1 we show the extended reasoning cycle originally introduced in [26] and first extended in [7] to include the *ProcAct* step.

3.1 Enriching Events in Safety Shields

A limitation of standard safety shields [16] is that the events being monitored are limited to the shielded plan. In other words, the execution trace observed by a safety shield can only contain events generated within the plan associated with that shield. In practice, this means that safety shields were unable to detect potential violations coming from other plans or other agents. Our extended $SelAppl'$ updates all shields across all current intentions in C_I. This allows shields to monitor not only their associated plans, but also events originating elsewhere in the agent or coming from other agents. As a result, a shield can depend on behaviour outside its own plan context, including events from non-shielded plans, since $SelAppl'$ imposes no restriction on te's source.

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

$$\begin{aligned} where \quad & C'_\rho = \langle p, \phi\rangle \\ & C'_I = \{\langle i, S'\rangle \mid \langle i, S\rangle \in C_I\} \\ & C'_{SusI} = \{\langle i, S'\rangle \mid \langle i, S\rangle \in C_{SusI}\} \\ & S' = \{\langle \sigma', \varphi, i'\rangle \mid \langle \sigma, \varphi, i'\rangle \in S \ \wedge \ \sigma' = \sigma \cdot te\} \end{aligned}$$

The events observed by the shield now come from multiple sources (not just the shield's intention). Thus, a violation could be caused by such external events to the shielded plan. This could be a problem because the shield does not have control over such events. A possible way to recognise this issue and at the same time tackle it, is to make the plan fail when the shield's violation is caused by an external event to the shielded plan, as done in [16]. On the other hand, when the event causing the shield violation is related to the shielded plan, the response to the violation can be more precise.

3.2 Suspending Intentions with Safety Shields

The following revised rule handles failures caused by a shield violation that occurs during the execution of a plan.

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

$$\begin{aligned} where \quad & C'_I = C_I \setminus \{\langle i, S\rangle\} \\ & C'_{SusI} = C_{SusI} \cup \{\langle i, S\rangle\} \end{aligned}$$

This situation is detected by evaluating whether the trace s_σ associated with a shield no longer satisfies its corresponding safety property s_φ, *i.e.*, $s_\sigma \not\models s_\varphi$. Importantly, this condition alone does not imply that the most recent event—typically the goal adoption $!at$—is solely responsible for the violation. While $!at$ may coincide with the moment the violation is detected, the violation may have been caused by previous or external events, given that shield traces can now be influenced by multiple sources. Therefore, attributing causality to $!at$ is only justifiable under the assumption that the trace was compliant until that event, which may not always hold. In cases where a violation is detected but cannot be clearly linked to the current plan's internal execution, recovery shields are required to attempt to deal with the violation (more details in the following section).

$$(RecoverSusI)\frac{}{C, RecSusI \rightarrow C', SelInt}$$

$$\begin{aligned} where \quad & C'_I = C_I \cup \{\langle i, S\rangle \mid \langle i, S\rangle \in C_{SusI} \ \wedge \\ & \qquad \nexists_{s \in S}.(s_\sigma \not\models s_\varphi)\} \\ & C'_{SusI} = C_{SusI} \setminus \{\langle i, S\rangle \mid \langle i, S\rangle \in C_{SusI} \ \wedge \\ & \qquad \nexists_{s \in S}.(s_\sigma \not\models s_\varphi)\} \end{aligned}$$

Once an intention is suspended due to a shield violation, it remains inactive until the associated shield condition is no longer violated. That is, when the execution trace s_σ is once again compliant with

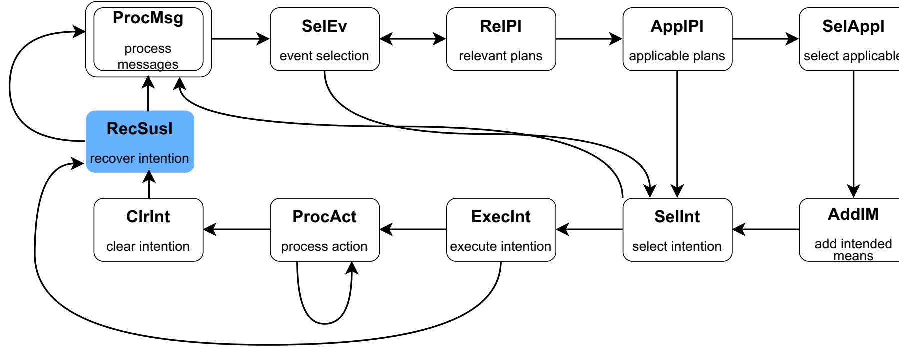

Figure 1: The extended AgentSpeak(L) reasoning cycle with the new `RecSusI` step, which checks suspended intentions against recovery shields and resumes them when conditions become safe.

the safety property s_φ. The reasoning cycle includes a dedicated step (*i.e.*, $RecoverSusI$) to periodically evaluate suspended intentions and resume them when their shields return to a valid state (as shown in Figure 1). This ensures that suspended intentions are not abandoned indefinitely and can continue execution once the environment permits safe compliance with their formal constraints.

Before presenting the technical details of recovery shields, we introduce an example that illustrates the type of violations our revised mechanism addresses through suspension.

Example 1 (Suspension due to internal error)**.** *Consider a plan that, upon execution, adds the belief b to the belief base and then proceeds to call a sub-goal $!g$. We want to enforce the property $\Box(+b \rightarrow \bigcirc(+c))$. $\Box$ and $\bigcirc$ are the LTL temporal operators for always (globally) and next. Whenever b is added to the belief base, the belief c must be added as well in the next step. Importantly, in this example, c is assumed to result from an external source (e.g., the environment or another agent). Suppose the agent begins executing the following trace:*

$$\sigma = +!t,\ +b,\ +!g$$

Here, '$+!t$' is the plan trigger, '$+b$' is an internal belief addition, and '$+!g$' is the intended next step. However, since $+c$ has not yet been observed, the shield detects a potential violation of the property $\Box(+b \rightarrow \bigcirc(+c))$. Instead of failing the intention, the shield suspends it, preventing the execution of '$+!g$', and thus preventing it from being added to the execution trace. Later, due to another plan or an external event, the belief $+c$ is added:

$$\sigma' = +!t,\ +a,\ suspend,\ +c$$

Once $+c$ is observed, the shield recognises that the recovery condition is now satisfied and resumes the suspended intention, allowing execution to continue with '$+!g$'.

This example illustrates how safety shields enable agents to handle temporary inconsistencies caused by environmental delays or concurrent activity, without prematurely abandoning goals. However, safety shields can only recover as long as the event that caused a violation is internal to the shielded plan. The reason for this is that the shielded plan can detect the violation and prevent the event from being executed. Therefore, the safety shield will suspend the intention of the shielded plan, but it will remain active while observing for external events that would satisfy the property and allow the intention to be resumed. To deal with events that cause a violation that are external to the shielded plan we need recovery shields.

3.3 Recovery Shields

With *safety shields*, we can constrain the behaviour within an agent's plans. However, we do not propose any specific strategy for handling situations where a safety shield is violated, other than suspending the intention responsible for the violation (rule $IntEvFail'$) and resuming it (rule $RecoverSusI$) once the safety shield returns to a safe state—*i.e.*, when the intention stack is no longer considered to be in violation.

Although this mechanism effectively ensures that intentions are only resumed when it is safe to do so, it may lead to significant delays or even prevent resumption entirely when resolving the violation requires conditions beyond simply observing the safety property. In such cases, rather than continuing to monitor the safety shield, we propose complementing it with what we term *recovery shields*. These do not constrain the agent's behaviour. Instead, they formally specify the conditions under which a suspended intention can safely be resumed, focusing on recovery properties rather than safety violations.

Unlike safety shields, recovery shields can be automatically synthesised. When a safety property is violated, a recovery shield can be generated to monitor whether the cause of the violation is eventually resolved. For example, if the violation was triggered by the addition of a particular belief, the recovery shield can track whether that belief is later removed. Similar mechanisms can be applied to other types of violations. In the following sections, we detail the synthesis process of recovery shields and explain how they influence the agent's reasoning cycle.

Definition 4. *A recovery shield is defined as a tuple $\langle\sigma, S, \psi\rangle$, where σ is a sequence of events (as in a standard safety shield), S is the set of safety shields associated with the suspended intention at the time of its suspension, and ψ is the formula that, if satisfied at runtime, triggers the resumption of the intention along with its associated safety shields S.*

As mentioned above, unlike a safety shield, a recovery shield is not added through annotation, but is automatically synthesised and added at runtime upon violating a safety shield.

Safety shields and recovery shields are not mutually exclusive when resuming a suspended intention. In fact, they can be used together, as they monitor different types of situations. Safety shields are intended for cases where the violation is caused internally by the intention's own execution, meaning the failure arises within the plan itself, not due to external interference, such as belief changes made by another agent or plan. In these cases, there is no need to generate a recovery shield, since no external condition must be observed. Instead, the safety shield continues monitoring and can resume the intention if a new event restores compliance.

On the other hand, when a violation is caused by an external event, for example, by a belief added by another plan or agent, then the event itself is not necessarily wrong, but leads to a temporary viola-

tion. Here, a recovery shield becomes useful, as it monitors for when the external cause is resolved and allows the suspended intention to continue safely. Therefore, safety and recovery shields complement each other by addressing different sources of failure.

3.3.1 Generating Recovery Shields

Upon the violation of a safety shield $\langle \sigma, \varphi, i \rangle$, where $\langle i, S \rangle \in C_I$, we must generate a corresponding recovery shield of the form $\langle \emptyset, S, \psi \rangle$[1]. To construct this recovery shield, we first synthesise the temporal goal ψ, which forms the core of the shield and determines whether the suspended intention i, which triggered the violation, can be safely resumed. To this end, we define a function, denoted as $GenRecGoal$, that generates the appropriate recovery goal. This temporal formula ψ must be satisfied in order to resume the suspended intention. The function is defined as follows.

$$
\begin{aligned}
&GenRecGoal(\sigma, \varphi) = \psi \\
&\qquad \text{where} \quad j = \min\{j \in \mathbb{N} \mid 1 \le j \le |\sigma| \ \wedge\ \sigma_{\le j} \not\models \varphi\} \\
&\qquad \text{and} \quad \psi = \bigwedge \{gen(\sigma[k]) \mid j \le k \le |\sigma|\}
\end{aligned}
$$

We first identify the minimum index j at which the trace σ begins to violate the safety property φ to determine the earliest point in the trace where φ is no longer satisfied. It is important to note that, since σ is updated during the reasoning cycle by all active intentions, multiple events may contribute to the violation of φ. In contrast, under a simplified scenario where only the intention associated with a shield can update σ, the violation would always be caused solely by the most recent event, since the intention would be suspended immediately upon violation, preventing any further updates.

Once the minimum index j has been determined, the temporal goal ψ is constructed as a conjunction of temporal formulas, each generated by the function gen. Specifically, for every event contributing to the violation—*i.e.*, every event between index j and the final event in σ—the function gen produces a corresponding subgoal aimed at restoring consistency with φ.

The function gen can be defined as follows ($\Diamond$ is the LTL temporal operator for *eventually*):

$$
\begin{aligned}
gen(+bel) &= \Diamond(-bel) \\
gen(-bel) &= \Diamond(+bel) \\
gen(+!goal) &= \Diamond(-!goal) \\
gen(-!goal) &= \Diamond(+!goal)
\end{aligned}
$$

Given a triggering event such as the addition or removal of a belief, or the addition or removal of a goal, the function gen produces a corresponding temporal goal, expressed as an LTL formula. This goal represents the desired condition to be *eventually* observed in order to “undo” the effect of the triggering event.

Correctness and Completeness. The $GenRecGoal$ function aims to produce a recovery condition ψ such that, if ψ holds, the violated property φ is re-established in the execution trace, ensuring *soundness*. Intuitively, ψ is derived by analysing the violation trace and synthesising a set of events whose occurrence restores φ. We do not provide guarantees of *completeness* (*i.e.*, a suitable ψ may not exist) nor of *minimality* (*i.e.*, ψ may include superfluous constraints). Formal proofs are omitted due to space constraints.

[1] The first item is $\emptyset$ because upon creation the recovery shield has not yet observed any new event, so its σ is empty.

3.3.2 Adding Recovery Shields

A recovery shield is added upon the violation of a safety shield. Thus, its addition is handled in the newly revised $IntEvFail''$ rule.

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

$$
\begin{aligned}
\textit{where} \quad & C'_I = C_I \setminus \{\langle i, S \rangle\} \\
& C'_{SusI} = C_{SusI} \cup \{\langle i, recoveryS \rangle\} \\
& recoveryS = \langle \emptyset, S, GenRecGoal(s_\sigma, s_\varphi) \rangle
\end{aligned}
$$

Differently from the standard and previously revised rule, $IntEvFail''$ handles the violation of a safety shield by generating a recovery shield ($recoveryS$) and by adding the latter to C_{SusI}.

3.3.3 Updating Recovery Shields

A recovery shield, like a safety shield, must be updated when events are selected during the agent's reasoning cycle. Therefore, we need to revise the rule as before to ensure that the newly added recovery shields are properly updated.

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

$$
\begin{aligned}
\textit{where} \quad & C'_\rho = \langle p, \phi \rangle \\
& C'_I = \{\langle i, S' \rangle \mid \langle i, S \rangle \in C_I\} \\
& C'_{SusI} = \{\langle i, rec' \rangle \mid \langle i, rec \rangle \in C_{SusI}\} \\
& S' = \{\langle \sigma', \varphi, i' \rangle \mid \langle \sigma, \varphi, i' \rangle \in S \ \wedge\ \sigma' = \sigma \cdot te\} \\
& rec' = \langle rec_\sigma \cdot te, rec_S, rec_\psi \rangle
\end{aligned}
$$

The revised rule operates on the updated version of C_{SusI}, which includes both the suspended intentions and their associated recovery shields. In this rule, we update not only the shields but also the recovery shields themselves, specifically by modifying their σ component and adding the observed triggering event te.

3.3.4 Removing Recovery Shields

We must also consider the removal of a recovery shield, which occurs when an intention can be resumed. Unlike the earlier approach based solely on safety shields (rule $RecoverSusI$), we can now enhance the resumption process by inspecting the recovery shields. This leads us to the following revised rule.

$$
{\scriptstyle (RecoverSusI')}\frac{}{C, RecSusI \to C', SelInt}
$$

$$
\begin{aligned}
\textit{where} \quad & C'_I = C_I \cup \{\langle i, S \rangle \mid \langle i, rec \rangle \in C_{SusI} \wedge \\
& \qquad\qquad rec_\sigma \models rec_\psi)\} \\
& S = \{\langle \sigma', \varphi, i \rangle | \langle \sigma, \varphi, i \rangle \in rec_S \wedge \\
& \qquad\qquad \sigma' = \sigma \cdot rec_\sigma\} \\
& C'_{SusI} = C_{SusI} \setminus \{\langle i, rec \rangle \mid \langle i, rec \rangle \in C_{SusI} \wedge \\
& \qquad\qquad rec_\sigma \models rec_\psi\}
\end{aligned}
$$

This rule checks, for each suspended intention, whether its associated recovery shield is currently satisfied (*i.e.*, whether all the necessary events to resume the intention have been observed). Each intention that meets this condition is reinserted into C_I, along with its corresponding set of safety shields S.

Resuming an intention, along with its attached safety shields, does not by itself guarantee that the agent has returned to a safe state. The condition that triggered the original suspension may no longer

hold, as indicated by the recovery shield, but other problematic situations may have arisen during the suspension. To ensure that safety shields can correctly assess the current context, their σ traces must be updated with the events observed by the recovery shield while the intention was suspended. This way, the shields gain a complete and up-to-date view of the execution trace, allowing them to detect any new violations. If such a violation is found, the intention can be suspended again, thus maintaining robust safety enforcement. This is achieved by appending the recovery shield's trace rec_σ to the σ of each shield in the set S attached to the resuming intention i, before the latter is resumed and new events may be observed.

Example 2 (Suspension due to external error)**.** *Recovery shields are designed to handle violations caused by external interference that disrupts the temporal constraints enforced by safety shields. When a violation occurs due to events beyond the agent's control, the intention is suspended and a recovery condition is generated to restore a consistent execution trace. Consider the property $\Box(+b \to \bigcirc(+!g))$, which states that whenever the belief b is added, the agent must perform the goal '$+!g$' in the next step. Now, suppose the agent begins executing a plan and adds '$+b$', but before it can proceed with '$+!g$', external actions inject unrelated events that interfere with the expected trace. Let us consider the following trace:*

$$\sigma = +!t,\ +b,\ +c,\ -d,\ suspend$$

Here, '$+!t$' is the plan trigger; '$+b$' is added by the plan; then '$+c$' and '$-d$' are added externally (e.g., by another agent or the environment), violating the safety property. The safety shield cannot allow the intention to proceed with '$+!g$' under this altered context. Rather than failing the plan, the intention is suspended, and a recovery shield is synthesised. The safety shield suspends the intention and triggers the generation of a recovery shield, which is automatically synthesised according to our framework. Based on the interfering events ('$+c$', '$-d$'), the recovery condition requires that their effects be reversed (i.e., observe '$-c$' and '$+d$'). Once these recovery events are observed, the condition is satisfied and the suspended intention resumes:

$$\sigma' = +!t,\ +b,\ +c,\ -d,\ suspend,\ -c,\ +d,\ resume,\ +!g$$

This example demonstrates how recovery shields allow agents to delay execution until interfering events are compensated for. The ability to recover from environmental or multi-agent interference is critical in open and concurrent systems, where unintended interleaving is common.

4 Implementation

We developed a prototype[2] using the JaCaMo framework [5]. Since directly extending Jason's reasoning cycle would require intrusive modifications to its source code, we adopt a modular strategy based on plan instrumentation. Our prototype interprets LTL formulas over traces of belief and goal events emitted through this instrumentation. This design provides generality and avoids assumptions about domain-specific actions. Nevertheless, the approach is extensible: traces could be augmented with actions, state snapshots, or other signals. However, recovery from action-level violations is domain-dependent as shown in [16] and beyond the scope of this work.

While Jason [6] provides the agent programming layer, JaCaMo offers a richer environment model through CArtAgO [23], which allows agents to perceive and act upon the environment via observable properties and operations exposed by artifacts. Although JaCaMo includes support for organisational reasoning, our work focuses on the individual agent level, leaving organisational-level robustness to related efforts [2, 1].

The reason for using JaCaMo rather than Jason alone is that CArtAgO artifacts are well-suited for implementing runtime shields. Each agent is associated with an artifact responsible for tracking shield information and mediating enforcement through operations such as adding, updating, and removing shields. LTL properties are monitored at runtime using LamaConv[3], a Java library for temporal logic monitoring.

All enforcement logic is handled by a dedicated CArtAgO artifact, which encapsulates both safety and recovery shields. Upon activation of a shielded plan, the artifact sets up a monitor for the associated safety property. As the agent executes, relevant events are forwarded to the artifact, which checks them against all active safety shields. When a violation occurs, the intention is suspended and, if the cause appears to originate from outside the plan itself, a recovery shield is synthesised automatically. This recovery shield specifies a condition that, when satisfied, allows the suspended intention to resume safely.

The artifact continues to monitor both suspended safety shields and recovery shields. If a suspended shield becomes satisfied again, or the condition of a recovery shield is met, the artifact signals the agent to resume the corresponding intention. In this way, plans may be suspended and resumed multiple times based on a precise account of both internal behaviour and external interference.

Our implementation relies on a pre-processing step where annotated AgentSpeak(L) plans are automatically transformed via a Python script. The script removes shield annotations and adds the necessary instrumentation code. We insert operations to initialise each plan with a shield, check its conditions before each command, and clean it up when the plan terminates. A plan such as:

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

is transformed into the following form:

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

The auxiliary plan for the goal `check_command` is defined as:

```
+!check_command(Cmd) <-
  update_shield(Cmd, Verdict);
  if (not(Verdict)) {
    .suspend;
    !check_command(Cmd);
  }.
```

Here, `add_shield` registers the monitor for the corresponding property when the plan is selected. Each command is checked in advance through `update_shield`, and if a violation is detected, the intention is suspended and the command is retried once conditions improve. The `remove_shield` operation then removes the shield once the plan concludes.

This design ensures shield logic remains fully decoupled from the agent's internal reasoning mechanisms. Agents remain autonomous, while the artifact transparently handles monitoring, enforcement, suspension, and resumption. The result is a lightweight and modular runtime enforcement mechanism that is both flexible and robust.

[2] https://github.com/AngeloFerrando/SafetyAndRecoveryShieldsBDI (Accessed: 24 July 2025)

[3] https://www.isp.uni-luebeck.de/lamaconv (Accessed: 30-April-2025)

Case Study: Personal Assistant. To illustrate our approach, we implemented a scenario involving three agents in a personal assistant multi-agent system: `human`, `assistant`, and `dentist`. The `human` agent initiates the process by requesting a booking at a specific time (e.g., 15:00), sending the goal to the `assistant`, who coordinates the interaction with the `dentist`.

The `assistant` maintains a belief base of currently blocked times and responds to booking requests via the following plan:

```
@shield(globally(not +blocked(Time)))
+!book_dentist(Time) :
blocked(ListTime) & not .member(Time,ListTime) <-
    .send(dentist, achieve, booking(15));
    .wait({+dentist_confirmation});
    .print("Booking successful!").
```

This plan is protected by a safety shield that enforces the temporal property that it must never be the case that `blocked(Time)` is added while the plan is running. This ensures that the booking process can only proceed if the selected time remains unblocked throughout.

The `human` agent initiates the booking and, partway through, simulates a conflicting commitment by telling the assistant that the time has become unavailable (the `.wait()` actions are added to wait a number of milliseconds to simulate a more realistic execution):

```
+!book_dentist(Time) <-
  .send(assistant, achieve, book_dentist(Time));
  .wait(1000);
  .send(assistant, tell, blocked(Time));
  .wait(5000);
  .send(assistant, untell, blocked(Time)).
```

This sequence reflects a real-world situation: the user initially intends to go to the dentist, but something else comes up, perhaps a friend suggests going to the cinema, so the time becomes blocked. The shield detects the addition of the `blocked(15)` belief, violating its property, and suspending the intention[4]. At the moment of suspension, a recovery shield is automatically synthesised with the property `eventually -blocked(Time)`, expressing that the intention may only resume once the belief `blocked(Time)` is removed. This ensures that resumption is not arbitrary but conditionally tied to the resolution of the original violation.

Later, the friend cancels, and the user no longer needs to block that time. When the `human` sends the `untell` message, the recovery shield detects that the issue has been resolved (*i.e.*, the `-blocked(Time)` is observed) and consequently the recovery shield is satisfied. The suspended intention is resumed and proceeds to complete the booking. The `dentist` agent, upon receiving the request, confirms the appointment:

```
+!booking(Time) <-
  .wait(1000);
  .send(assistant, tell, dentist_confirmation).
```

While the above scenario focuses on a simple temporal property, our framework can also capture richer constraints that combine temporal and contextual conditions. For instance, consider a plan to book a blood glucose test at 10:00, which requires fasting. A possible property for this plan could state that: (i) the agent must not believe or intend to book breakfast before 11:00; (ii) the interval around 10:00 remains free of conflicting appointments; and (iii) no prior appointment should break the fasting requirement. Such a property can be formulated in LTL over belief and goal events as:

```
@shield(globally(not (+!book_breakfast | +blocked
    (10))) & (not +appointment(_) until +!
    book_test(10)))
```

This illustrates how domain-specific safety requirements can be encoded using our event-based monitoring approach.

5 Related Work

Failure handling has long been a central theme in BDI systems. Classic approaches include retrying or replacing failed plans [18, 24], or defining recovery plans explicitly [7]. Some approaches exploit concurrent intentions to resolve failures [28]. Recent architectural proposals support rapid fail-safe behaviours by enabling agents to temporarily switch reasoning modes in response to critical situations [4]. While effective, these strategies still require developers to script recovery behaviour. By contrast, our approach automates both detection and recovery through the use of runtime monitors and synthesised recovery conditions.

Other recovery mechanisms have been explored in organisational multi-agent systems. A recent accountability-based framework records which agent is responsible for a failed goal or interaction, enabling reassignment or exception logging [2]. While well-suited to team coordination, these approaches assume organisational structures and external oversight. Our method remains fully decentralised, operating within the agent's local reasoning cycle through declarative, agent-specific recovery conditions.

Plan execution strategies also impact agent robustness. Recent work shows that delaying variable binding during plan selection improves adaptability in dynamic environments, particularly when paired with recovery mechanisms [25]. This could complement recovery shields, which resume suspended intentions only once their associated safety conditions are re-established.

Our approach also relates to maintenance goals [13, 14], which preserve desired conditions over time. However, maintenance goals rely on predefined handler plans, while shields enforce temporal properties declaratively. Recovery shields extend this further by automatically synthesising the conditions for safe resumption, removing the need for explicit recovery logic.

6 Conclusions and Future Work

We introduced an extension to AgentSpeak(L) that enables runtime enforcement of safety properties via intention-level shields. Unlike prior work that fails plans immediately upon violation [16], our approach suspends violating intentions and resumes them once safe execution is possible, improving flexibility and robustness in multi-agent settings with internal and external disruptions.

A current limitation is the assumption that agents have full observability of the relevant events required for shield evaluation. Extending the framework to handle partial observability would increase applicability in more realistic environments. Recent work in RV under partial observability [17] shows that monitoring can be adapted to scenarios with incomplete information. Building on these ideas, techniques such as inferred events or belief-based abstractions—widely used in multi-agent reasoning—could be integrated into our approach to approximate unobservable traces. Future work also includes supporting shared or synchronised shields across agents, which would enable runtime enforcement of global properties, raising interesting challenges around trace aggregation, coordination, and distributed recovery.

[4] Although the shield is specified parametrically using `Time`, the variable is grounded at runtime when the plan is selected. As a result, the actual event observed (e.g., `+blocked(15)`) is matched against the concrete, instantiated property (e.g., `globally(not +blocked(15))`).

References

[1] M. Baldoni, C. Baroglio, O. Boissier, R. Micalizio, and S. Tedeschi. Distributing responsibilities for exception handling in JaCaMo. In *20th International Conference on Autonomous Agents and Multiagent Systems*, pages 1752–1754. ACM, 2021. doi: 10.5555/3463952.3464226.

[2] M. Baldoni, C. Baroglio, R. Micalizio, and S. Tedeschi. Accountability in multi-agent organizations: from conceptual design to agent programming. *Auton. Agents Multi Agent Syst.*, 37(1):7, 2023. doi: 10.1007/s10458-022-09590-6.

[3] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, Cham, 2018. doi: 10.1007/978-3-319-75632-5_1.

[4] L. B. Becker, I. de Oliveira Silvestre, J. F. Hübner, and M. Fisher. An expedited BDI agent architecture: Improving the responsiveness of agent-based autonomous systems for handling critical situations. *Robotics Auton. Syst.*, 186:104917, 2025. doi: 10.1016/J.ROBOT.2025.104917.

[5] O. Boissier, R. 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. ISBN 9780262360661.

[6] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*, volume 8. John Wiley & Sons, Ltd, United Kingdom, 10 2007. doi: 10.1002/9780470061848.

[7] R. H. Bordini and J. F. 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*, pages 635–640. IOS Press, 2010. doi: 10.3233/978-1-60750-606-5-635.

[8] R. H. Bordini, A. E. F. Seghrouchni, K. V. Hindriks, B. Logan, and A. Ricci. Agent programming in the cognitive era. *Auton. Agents Multi Agent Syst.*, 34(2):37, 2020. doi: 10.1007/s10458-020-09453-y.

[9] R. C. Cardoso and A. Ferrando. A review of agent-based programming for multi-agent systems. *Computers*, 10(2):16, Jan 2021. ISSN 2073-431X. doi: 10.3390/computers10020016.

[10] S. Cranefield, M. Winikoff, V. Dignum, and F. Dignum. No pizza for you: Value-based plan selection in BDI agents. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 178–184. IJCAI, 2017. doi: 10.24963/ijcai.2017/26.

[11] L. de Silva, F. Meneguzzi, and B. Logan. BDI agent architectures: A survey. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4914–4921. ijcai.org, 2020. doi: 10.24963/IJCAI.2020/684.

[12] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini. Model checking agent programming languages. *Autom. Softw. Eng.*, 19(1):5–63, 2012. doi: 10.1007/s10515-011-0088-x.

[13] S. Duff, J. Harland, and J. Thangarajah. On proactivity and maintenance goals. In *5th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1033–1040. ACM, 2006. doi: 10.1145/1160633.1160817.

[14] S. Duff, J. Thangarajah, and J. Harland. Maintenance goals in intelligent agents. *Comput. Intell.*, 30(1):71–114, 2014. doi: 10.1111/coin.12000.

[15] Y. Falcone, L. Mounier, J. Fernandez, and J. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods Syst. Des.*, 38(3):223–262, 2011. doi: 10.1007/s10703-011-0114-4.

[16] A. Ferrando and R. C. Cardoso. Failure handling in BDI plans via runtime enforcement. In *ECAI 2023 - 26th European Conference on Artificial Intelligence, Kraków, Poland*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 716–723. IOS Press, 2023. doi: 10.3233/FAIA230336.

[17] A. Ferrando and V. Malvone. Runtime verification with imperfect information through indistinguishability relations. In *Software Engineering and Formal Methods - 20th International Conference*, volume 13550 of *Lecture Notes in Computer Science*, pages 335–351. Springer, 2022. doi: 10.1007/978-3-031-17108-6_21.

[18] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning and Acting*. Cambridge University Press, Cambridge, UK, 2016. ISBN 978-1-107-03727-4.

[19] A. Lomuscio, H. Qu, and F. 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. doi: 10.1007/s10009-015-0378-x.

[20] A. Pnueli. The temporal logic of programs. In *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57. IEEE Computer Society, 1977. doi: 10.1109/SFCS.1977.32.

[21] A. 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*, pages 42–55. Springer, 1996. doi: 10.1007/BFb0031845.

[22] A. S. Rao and M. 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*, pages 312–319. The MIT Press, 1995.

[23] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment programming in CArtAgO. In *Multi-Agent Programming: Languages, Tools and Applications*, Multiagent Systems, Artificial Societies, and Simulated Organizations, chapter 8, pages 259–288. Springer, Boston, MA, 2009. ISBN 978-0-387-89298-6. doi: 10.1007/978-0-387-89299-3_8.

[24] S. Sardiña and L. Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Auton. Agents Multi Agent Syst.*, 23(1):18–70, 2011. doi: 10.1007/s10458-010-9130-9.

[25] F. Vidensky, F. Zboril, J. Beran, R. Kocí, and F. V. Zboril. Comparing variable handling strategies in BDI agents: Experimental study. In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence, ICAART 2024*, pages 25–36. SCITEPRESS, 2024. doi: 10.5220/0012358600003636.

[26] R. Vieira, Á. F. Moreira, M. J. Wooldridge, and R. 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. doi: 10.1613/jair.2221.

[27] M. 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*, pages 251–259. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3091166.

[28] Y. Yao, B. Logan, and J. 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*, pages 2558–2565. AAAI Press, 2016. doi: 10.1609/aaai.v30i1.10129.