Safety Shields, an Automated Failure Handling Mechanism for BDI Agents

Angelo Ferrando
University of Genova
Genova, Italy
angelo.ferrando@unige.it

Rafael C. Cardoso
The University of Manchester
Manchester, United Kingdom
rafael.cardoso@manchester.ac.uk

ABSTRACT
The Belief Desire Intention model is a widely used architecture for developing rational agents. Because of its expressiveness, the task of programming a BDI agent can be challenging, especially when applied to safety-critical scenarios. In such scenarios, it is important to provide a safeguard for the critical behaviour of the agent. In this paper, we summarise how to extend the agent’s reasoning cycle in the BDI model with safety shields. A safety shield works as a sandbox for the agents’ plans that is enforced at runtime so that the agent behaves according to a safety formal specification. A runtime monitor is automatically synthesised from a shield to detect any failure that is within the scope of the shielded plan.

KEYWORDS
failure handling; BDI; Multi-Agent Systems; runtime verification

1 INTRODUCTION
Engineering a software system can be a complex process. This is especially true when the system under consideration presents some degree of autonomy. In the context of Multi-Agent Systems (MAS), multiple entities called agents are programmed and deployed in a distributed fashion to solve various types of tasks.

In this paper, we consider MAS designed and developed following the principles of the Belief Desire Intention (BDI) model [13]. We choose the BDI model because it is one of the most popular architectures for the development of MAS. The BDI model is part of the symbolic approaches to Artificial Intelligence (AI) development, hence it expects the developer to fully specify how an agent behaves. This is obtained by defining, beliefs, goals, and especially plans, which denote – step by step – 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) [12], are notoriously different from traditional programming languages and usually come with a steep learning curve. The process of testing [15], debugging [16], and verifying [9], such systems can be quite complex. When these BDI languages are applied to safety-critical scenarios, in which an error can be costly, any solution which may make the BDI development more reliable is of utmost importance.

The main idea of this work is to use Runtime Verification (RV) [2, 11] as a way to enforce safety properties [1] on BDI agents. These properties can only be violated at runtime, which means the resulting monitor can only report negative and inconclusive verdicts. This is due to the fact that safety properties are satisfied only by infinite traces of events, and at runtime we only have access to finite traces. BDI agents can be applied to dynamic scenarios, where it may be difficult to guarantee that their behaviour will always be consistent with the developers’ expectations. Runtime verification is usually more focused on detecting unexpected behaviours, rather than enforcing the system to actually behave in a correct way. Enforcing a behaviour leads to Runtime Enforcement [10].

We synthesise runtime monitors (called safety shields) to enforce the correct behaviour of existing BDI agents. In this paper, we summarise the main features of such safety shields, along with their generation and integration into the BDI architecture. A safety shield works as a sandbox for the agent. Every command (actions, addition/removal of beliefs, and so on) 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.

2 AGENTSPEAK(L) OPERATIONAL SEMANTICS
An AgentSpeak(L) configuration $C$ is a tuple $\langle I, E, A, R, Ap, i, p, \epsilon \rangle$ where $I$ is the set of intentions $\{i, i', \ldots\}$. Each intention $i$ is a stack of partially instantiated plans $\{p1, p2, \ldots, pn\}$. We use the $|$ symbol to separate plans in an intention. $E$ is a set of events $\{(\tau, i), (\tau', i'), \ldots\}$. Each event is a pair $\langle \tau, i \rangle$, where $\tau$ is a triggering event and the intention $i$ are plans associated with $\tau$. $A$ is a set of actions $\{(a, i), (a', i'), \ldots\}$. Each event is a pair $\langle a, i \rangle$, where $a$ is an action and the intention $i$ are plans associated with $a$. $R$ is a set of relevant plans. $Ap$ is a set of applicable plans. $\tau$, $\epsilon$ and $p$ keep the record of a particular intention, event and applicable plan (respectively) being considered in the current agent’s reasoning cycle. This notation is similar to the ones presented in [6, 12, 14].

To keep the notation compact, we adopt the following notations: (i) if $C$ is an AgentSpeak(L) configuration, we write $C_E$ to make reference to the component $E$ of $C$ (same for the other components of $C$); (ii) we write $C_i = \ldots$ to indicate there is no intention considered in the agent’s execution (same for $C_A$ and $C_E$); (iii) we write $i[p]$ to denote the intention that has $p$ on its top.
3 SAFETY SHIELDS

In this section, we introduce the notion of safety shields for the BDI model. Specifically, we extend the standard AgentSpeak(L) operational semantics (i.e., the inference rules). Due to space constraints, we present only some of the rules that need to be extended.

A shield is a component which can be attached to an agent’s plan to check whether the triggering event violates a formal specification during its execution. In such case, the shield enforces the plan to conform.

Safety Shield Specification. The first aspect to tackle is how, and when, a safety shield is specified. We achieve this by annotating the plans which we want to “shield”. Annotating plans is a common practice in existing BDI programming languages and can be found for example in [5, 8]. An annotation is a structured label attached to a plan. More formally, a shield annotation can be specified as follows: @shield[φ₁, …, φₙ] with (n ≥ 1) where shield is a custom label to identify that a shield annotation is being added, and φᵢ (with 1 ≤ i ≤ n) is the formal property the shield will look out for. 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.

Adding and Removing Safety Shields. This is obtained by extending the inference rules in the agent’s reasoning cycle. First, we have to consider where the shields are stored. Since each intention is attached to a certain plan and each plan is executed as an intention, then a shield can be attached to such intention. Thus, the shield is used to analyse events concerning the corresponding intention.

Capturing Violations (Failure Detection). Since the entire agent’s reasoning cycle depends on which plans are selected as relevant and, consequently, applicable. One possible way to enforce the satisfaction of a formal property is by extending the standard RelPlans function. The goal of such an extension is to take a property into consideration while selecting the relevant plans for a triggering event. The updated version is as follows: RelPlans(plans, τe, S)  

{ p | p ∈ plans ∧ σ = mgu(τe, TE(p)) ∧  \exists s ∈ S. σ → · te } \not∈ s_p  

where S denotes the set of shields associated to the current selected intention, and · denotes the concatenation amongst trace of events. In this way, we can check whether the triggering event τe violates at least one shield s in S (with s_p the trace observed up to now by s, and s_p the property checked by s). If that is the case, RelPlans returns the empty set.

Besides updating the RelPlans function, we also need to update the corresponding rule that makes use of it in the operational semantics. In particular the Rel₁ rule, which is defined as follows:  

\begin{align*}
\text{(Rel₁)} & : \quad \forall C, \text{beliefs} \rightarrow C', \text{beliefs} \quad C_e = \langle \tau e, i \rangle, C_A p = C_R = \emptyset \\
& \text{where} \quad C_R = \text{RelPlans}(\text{plans}, \tau e) \\
\end{align*}

Rel₁ takes the current event in C_e, and extracts the set of relevant plans for the specific triggering event τe. Its extension, which uses the new version of RelPlans, is defined as follows:  

\begin{align*}
\text{(Rel₃)} & : \quad \forall C, \text{beliefs} \rightarrow C', \text{beliefs} \quad C_e = \langle \tau e, i \rangle, C_A p = C_R = \emptyset, (i,S) < C_I \\
& \text{where} \quad C_I = \text{RelPlans}(\text{plans}, \tau e, S) \\
& C_I' = (C_I \setminus \{(i,S)\}) \cup \{(i,S')\} \\
& S' = \{(s',φ, i') \mid (s,φ, i) \in S ∧ s' = σ \cdot \tau e\} \\
\end{align*}

The updated rule is necessary to keep track of the events into S’s shields. Each time an event is considered in the agent’s reasoning cycle, it is also stored in every active shield in S for the corresponding intention i, to be evaluated in future executions.

Note that, when the triggering event (τe) violates at least one shield in S, RelPlans returns the empty set. Thus, no relevant plan is available (C_R = 0), as well as no applicable plan (C_A p = 0); since AppPlans is defined on top of RelPlans. 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).

\begin{align*}
\text{(App₁)} & : \quad \langle \text{App}\{C_R, \text{beliefs}\} = 0, C_e = \langle \tau e, i \rangle, C_A p = 0, C_R = 0 \rangle \\
& \text{where} \quad C_E = \begin{cases} 
C_E \cup \{\langle ¬ \%\text{at}, i \rangle \} & \text{if } \tau e = \%\text{at with } % \in \{1,?\} \\
C_E \cup \{C_I\} & \text{otherwise} 
\end{cases}
\end{align*}

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.

4 IMPLEMENTATION

As a proof of concept, we implemented a prototype\(^2\) in the JaCaMo multi-agent development framework [3, 4]. Jason [5], which is the implementation of AgentSpeak(L) used in JaCaMo, is one of the most used and well-known BDI programming languages [7].

Specifically, we decided to use JaCaMo instead of Jason, because the former supports artifacts which are well-suited for implementing the shields and interfacing with the monitors. Artifacts allow agents to have better control over their shields, while in Jason this would have to be done in a shared Java environment. The artifact maintains all the information on the shields, and it is the object consulted when a shield needs to be added, removed, or updated.

5 CONCLUSIONS AND FUTURE WORK

In this extended abstract, we summarise the design and implementation of safety shields for BDI agents. We formally specify how to enhance the agent’s reasoning cycle to enforce the satisfaction of safety properties through shields. Some resulting extended inference rules are reported. The contribution is not only theoretical, but it comprises a practical component as well. A prototype of our approach is proposed, along with its integration in the JaCaMo platform.

For future work, we are interested in improving the integration in JaCaMo. The current implementation is based on instrumentation, which is a less invasive way to approach the problem at the implementation level. 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 (as proposed in the theory of this work). Also on the implementation side, we are interested in extending the work from using one single artifact per agent, to one artifact per shield. This extension should bring to better performances, above all in the case of nested shields.

\(^2\)https://github.com/AngeloFerrando/SafetyShieldsBDI
REFERENCES