Safety Shields, an Automated Failure Handling Mechanism for BDI Agents

Extended Abstract

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

ACM Reference Format:
Angelo Ferrando and Rafael C. Cardoso. 2022. Safety Shields, an Automated Failure Handling Mechanism for BDI Agents: Extended Abstract. In *Proc. of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2022), Online, May 9–13, 2022*, IFAAMAS, 3 pages.

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 uttermost 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 $\langle I, E, A, R, Ap, \iota, \rho, \epsilon \rangle$ where: I is the set of intentions $\{i, i', \ldots\}$. Each intention i is a stack of partially instantiated plans $[p_1|p_2 \ldots p_n]$. We use the | symbol to separate plans in an intention. 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 the intention i are plans associated with te. A is a set of actions $\{\langle a, i\rangle, \langle a', i'\rangle, \ldots\}$. 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. 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. 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 $C_\iota = \_$ to indicate there is no intention considered in the agent's execution (same for C_ρ and C_ϵ); (iii) we write $i[p]$ to denote the intention that has p on its top.

Proc. of the 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2022), P. Faliszewski, V. Mascardi, C. Pelachaud, M.E. Taylor (eds.), May 9–13, 2022, Online.

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 such plan 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: $@shield[\varphi_1, \ldots, \varphi_n]$ with $(n \geq 1)$ where *shield* is a custom label to identify that a shield annotation is being added, and φ_i (with $1 \leq i \leq n$) 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 shield 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.

Catching Violations (Failure Detection). Since the entire agent's reasoning cycle depends on which plans are selected as relevant[1] 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:

$$RelPlans(plans, te, S) =$$
$$\{p\sigma \mid p \in plans \ \wedge \ \sigma = mgu(te, TE(p)) \ \wedge \ \nexists_{s \in S}.s_\sigma \cdot te \not\models s_\varphi\}$$

where S denotes the set of shields associated to the current selected intention, and $\cdot$ denotes the concatenation amongst trace of events. In this way, we can check whether the triggering event te violates at least one shield s in S (with s_σ the trace observed up to now by s, and s_φ 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 Rel_1 rule, which is defined as follows:

$$(Rel_1)\frac{RelPlans(plans, te) \neq \emptyset}{C, beliefs \rightarrow C', beliefs} \; C_\epsilon{=}\langle te, i\rangle,\, C_{Ap}{=}C_R{=}\emptyset$$
$$where \qquad C'_R = RelPlans(plans, te)$$

Rel_1 takes the current event in C_ϵ, and extracts the set of relevant plans for the specific triggering event te. Its extension, which uses the new version of *RelPlans*, is defined as follows:

$$(Rel_{1'})\frac{RelPlans(plans, te, S) \neq \emptyset}{C, beliefs \rightarrow C', beliefs} \; C_\epsilon{=}\langle te, i\rangle,\, C_{Ap}{=}C_R{=}\emptyset,\, \langle i, S\rangle{\in}C_I$$
$$where \qquad C'_R = RelPlans(plans, te, S)$$
$$C'_I = (C_I \ \setminus \ \{\langle i, S\rangle\}) \ \cup \ \{\langle i, S'\rangle\}$$
$$S' = \{\langle \sigma', \varphi, i'\rangle \mid \langle \sigma, \varphi, i'\rangle \in S \ \wedge \ \sigma' = \sigma \cdot te\}$$

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 (te) violates at least one shield in S, *RelPlans* returns the empty set. Thus, no relevant plan is available ($C_R = \emptyset$), as well as no applicable plan ($C_{Ap} = \emptyset$); 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 ($-\%at$).

$$(Appl)\frac{AppPlans(C_R, beliefs) = \emptyset}{C, beliefs \rightarrow C', beliefs} \; C_\epsilon{=}\langle te, i\rangle,\, C_{Ap}{=}\emptyset,\, C_R{\neq}\emptyset$$
$$where \quad C'_E = \begin{cases} C_E \cup \{\langle -\%at, i\rangle\} & \text{if } te = +\%at \text{ with } \% \in \{!, ?\} \\ C_E \cup \{C_\epsilon\} & \text{otherwise} \end{cases}$$

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.

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 allows 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.

[1] A plan is relevant for a triggering event if the triggering event can successfully be unified with the plan's head.

[2] https://github.com/AngeloFerrando/SafetyShieldsBDI

REFERENCES

[1] Bowen Alpern and Fred B. Schneider. 1987. Recognizing Safety and Liveness. *Distributed Comput.* 2, 3 (1987), 117–126. https://doi.org/10.1007/BF01782772

[2] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). Lecture Notes in Computer Science, Vol. 10457. Springer, Cham, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[3] O. Boissier, R.H. Bordini, J. Hubner, and A. Ricci. 2020. *Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo.* MIT Press, United States. https://books.google.com.br/books?id=GM_tDwAAQBAJ

[4] Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* 78, 6 (Jun 2013), 747–761. https://doi.org/10.1016/j.scico.2011.10.004

[5] Rafael Bordini, Jomi Hübner, and Michael Wooldridge. 2007. *Programming Multi-Agent Systems in AgentSpeak Using Jason.* Vol. 8. John Wiley & Sons, Ltd, United Kingdom. https://doi.org/10.1002/9780470061848

[6] Rafael H. Bordini and Jomi Fred Hübner. 2010. Semantics for the Jason Variant of AgentSpeak (Plan Failure and some Internal Actions). In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings (Frontiers in Artificial Intelligence and Applications, Vol. 215)*, Helder Coelho, Rudi Studer, and Michael J. Wooldridge (Eds.). IOS Press, NLD, 635–640. https://doi.org/10.3233/978-1-60750-606-5-635

[7] Rafael C. Cardoso and Angelo Ferrando. 2021. A Review of Agent-Based Programming for Multi-Agent Systems. *Computers* 10, 2 (Jan 2021), 16. https://doi.org/10.3390/computers10020016

[8] Stephen Cranefield, Michael Winikoff, Virginia Dignum, and Frank Dignum. 2017. No Pizza for You: Value-based Plan Selection in BDI Agents. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, United States, 178–184. https://doi.org/10.24963/ijcai.2017/26

[9] Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini. 2012. Model checking agent programming languages. *Autom. Softw. Eng.* 19, 1 (2012), 5–63. https://doi.org/10.1007/s10515-011-0088-x

[10] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. 2011. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods Syst. Des.* 38, 3 (2011), 223–262. https://doi.org/10.1007/s10703-011-0114-4

[11] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebraic Methods Program.* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[12] Anand S. Rao. 1996. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1038)*, Walter Van de Velde and John W. Perram (Eds.). Springer, Berlin, Heidelberg, 42–55. https://doi.org/10.1007/BFb0031845

[13] Anand S. Rao and Michael P. Georgeff. 1995. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, Victor R. Lesser and Les Gasser (Eds.). The MIT Press, United States, 312–319.

[14] Renata Vieira, Álvaro F. Moreira, Michael J. Wooldridge, and Rafael H. Bordini. 2007. On the Formal Semantics of Speech-Act Based Communication in an Agent-Oriented Programming Language. *J. Artif. Intell. Res.* 29 (2007), 221–267. https://doi.org/10.1613/jair.2221

[15] Michael Winikoff. 2017. BDI agent testability revisited. *Auton. Agents Multi Agent Syst.* 31, 5 (2017), 1094–1132. https://doi.org/10.1007/s10458-016-9356-2

[16] Michael Winikoff. 2017. 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*, Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee (Eds.). ACM, Richland, SC, 251–259. http://dl.acm.org/citation.cfm?id=3091166