




# CAN-VERIFY: A Verification Tool For BDI Agents

Mengwei Xu[1(✉)], Thibault Rivoalen[3], Blair Archibald[2], and Michele Sevegnani[2]

[1] Department of Computer Science, University of Manchester, Manchester, UK
mengwei.xu@manchester.ac.uk
[2] School of Computing Science, University of Glasgow, Glasgow, UK
[3] Ecole Nationale de L'Aviation Civile (ENAC), University of Toulouse, Toulouse, France

**Abstract.** CAN-VERIFY is an automated tool that aids the development, verification, and analysis of BDI agents written in the Conceptual Agent Notation (CAN) language. It does not require users to be familiar with verification techniques. CAN-VERIFY supports syntactic error checking, interpretation of programs (running agents), and exhaustive exploration of all possible executions (agent verification and analysis) to check against both generic agent requirements, such as if a task can be achieved successfully, and user-defined requirements, such as whether a certain belief eventually holds. Simple examples of Unmanned Aerial Vehicles (UAV) and autonomous patrol robots illustrate the tool in action.

## 1 Introduction

The BDI architecture, where agents are modelled based on their beliefs, desires, and intentions, is a practical approach to developing complex and autonomous systems. BDI agents make decisions based on their beliefs, which reflect their understanding of the world, their desires, which represent their goals, and their intentions, which are the plans they've committed to achieve those goals. The design of these agents is challenging because of the mixed nature of the goal: declarative (a description of the state sought) and procedural (a set of instructions to perform), failure handling, inherently interleaved concurrent behaviours, and ultimately the safety of these agents' employment.

There is a software collection to develop BDI agents including JACK [33], Jason [10], and Jadex [28]. These platforms focus on the simulations of BDI agents primarily programmed in variants of AgentSpeak [29], one of the most well-known BDI languages. Simulations cannot, by their nature, analyse all possible agent behaviours. As such, formal verification techniques [20] have been used to exhaustively assess whether agents will indeed behave as required. For example, the Model Checking Agent Programming Languages (MCAPL) framework [16] offers model checking for GWENDOLEN [15] programs (another variant of AgentSpeak).


P. Herber and A. Wijs (Eds.): iFM 2023, LNCS 14300, pp. 364–373, 2024.
https://doi.org/10.1007/978-3-031-47705-8_19

To widen access to formal methods, we present an automated tool, CAN-VERIFY, for BDI programmers to verify agents in the Conceptual Agent Notation (CAN) language. As a superset of AgentSpeak, CAN includes advanced behaviours *e.g.* declarative goals, concurrency, and failure recovery. In a nutshell, CAN-VERIFY takes as input CAN programs and supports the following features:

- static checking for a wide range of BDI program syntactic issues;
- support for BDI program interpretation;
- exhaustive exploration of BDI program executions;
- verification, through model checking, of agents against a set of built-in generic agent requirements and (optional) user-defined agent requirements that can be expressed in natural language;
- verification of BDI agents parameterised by their initial belief base to support analysis of agent behaviours under different initial environments.

This is the first tool to support reasoning about CAN programs without requiring users to have specialised knowledge of verification techniques and formal logics. CAN-VERIFY, and all examples shown in this work, are openly available [35].

***Related Work.*** There is wide interest in applying formal verification to autonomous agents [7,12,25], but adoption has been limited mainly due to the complexities of formal verification tools [1]. This is also an issue in the broader Formal Method community for applicable formal methods [19]. Our tool allows agent programmers to benefit from verification without specialist knowledge.

Previous work on reasoning about BDI agents using automated techniques includes the MCAPL framework. MCAPL implements a BDI language in Java allowing it to be verified with the Java PathFinder [11] program model checker. While verifying an implementation tells you how the specific *system* will operate, it may not give insight into how the *language* semantics were meant to operate, especially if the implementation does not fully correspond. For example, GWENDOLEN (supported by MCAPL) only selects the first applicable plan, while language semantics usually allow *any* applicable plan to be chosen. Bordini et al. [9] translates a simplified AgentSpeak language into Promela [23] and verifies agents using the Spin model-checker [22]. Similarly, term-rewriting, specifically in Maude [14], has also been used to encode BDI agent languages, allowing verification of temporal properties with the Maude LTL model checker [17]. Recently, Jensen [24] applied the Isabelle/HOL proof assistant [27] to BDI programs. However, no tool is fully automated to relieve users from the error-prone translation and complex verification stages, whereas we streamline the BDI agent design, interpretation, and verification process, alleviating these burdens from users.

## 2 CAN—Overview

The Conceptual Agent Notation (CAN) [31] language formalises a classical BDI agent consisting of a belief base $\mathcal{B}$ and a plan library $\Pi$. The belief base $\mathcal{B}$ is a set

**Listing 1.1.** CAN agent for conference trip arrangement.

```
1 at_home // Initial belief bases
2 travel  // External events
3 // Plan library
4 travel: own_car & driving_distance <- start_car; driving.
5 // Action descriptions
6 start_car: car_functional <- <{engine_off}, {engine_on}>
7 driving: engine_on <- <{not_at_venue}, {at_venue}>
```

of formulae encoding the current beliefs and has belief operators for entailment (*i.e.* $\mathcal{B} \models \varphi$), and belief atom addition (resp. deletion) $\mathcal{B} \cup \{b\}$ (resp. $\mathcal{B} \setminus \{b\}$). In this case, we assume propositional logic. A plan library $\Pi$ is a collection of plans of the form $e : \varphi \leftarrow P$ with $e$ the triggering event, $\varphi$ the context condition, and $P$ the plan-body. The triggering event $e$ specifies why the plan is triggered, while the context condition $\varphi$ determines if the plan-body $P$ is able to handle the event. Events can either be external (*i.e.* from the environment) or internal (*i.e.* sub-goals that the agent itself tries to accomplish). The language used in the plan-body for agent programmers is $act \mid e \mid P_1; P_2 \mid P_1 \parallel P_2 \mid goal(\varphi_s, e, \varphi_f)$ where $act$ is an action, $e$ is a sub-event (*i.e.* internal event), composite programs $P_1; P_2$ (resp. $P_1 \parallel P_2$) for sequence (resp. concurrency), and $goal(\varphi_s, e, \varphi_f)$ a declarative goal program. Actions $act$ take the form $act : \varphi \leftarrow \langle \phi^-, \phi^+ \rangle$, where $\varphi$ is the pre-condition, and $\phi^-$ and $\phi^+$ are the deletion and addition sets of belief atoms, *i.e.* a belief base $\mathcal{B}$ is revised to be $(\mathcal{B} \setminus \phi^-) \cup \phi^+$ when the action executes. Composite programs $P_1; P_2$ is for sequenced execution and $P_1 \parallel P_2$ for interleaved concurrency. A declarative goal program $goal(\varphi_s, e, \varphi_f)$ expresses that the declarative goal $\varphi_s$ should be achieved through an event $e$, failing if $\varphi_f$ becomes true, and retrying as long as neither $\varphi_s$ nor $\varphi_f$ is true. The full semantics of CAN can be found in [30,34].

The example in Listing 1.1 describes a fragment of agent arranging a conference trip [3]. An agent desires to travel, *i.e.* an external event `travel` (line 2) and believes it is at home initially, *i.e.* a belief `at_home` (line 1). In this fragment, the only plan (line 4) expresses that if the agent believes it owns a car (*i.e.* `own_car` holds) and the venue is in driving distance (*i.e.* `driving_distance`), it will start the car (`start_car`) and drive (`driving`) to the venue. Actions change beliefs, for example, `start_car` (line 6) says that if the car is functional (`car_functional`) then, after executing the action, the agent should believe the engine is on (`engine_on`) and remove the belief the engine is off (`engine_off`).

## 3 CAN-VERIFY Components and Features

Our tool is based on an executable semantics of CAN [3] enabled through an encoding into Milner's Bigraphs [26], a computational model based on graph-rewriting. The dataflow of the toolchain is in Fig. 1. Step ① translates input CAN programs into bigraphs expressed in the BigraphER [32] language. During the translation, static checks are performed and errors/warnings reported to users.

**Listing 1.2.** Built-in (lines 1-4) and user-defined (lines 5 and 6) agent requirements. Belief-lists have $\wedge$ semantics *i.e.* there is a state where all beliefs hold at the same time.

```
1 A [ F ("no_failure" & X ("empty_intention")) ];
2 E [ F ("no_failure" & X ("empty_intention")) ];
3 A [ F ("failure" & X ("empty_intention")) ];
4 E [ F ("failure" & X ("empty_intention")) ];
5 A [ F (<belief-list>) ];
6 E [ F (<belief-list>) ];
```

To verify an agent, the agent requirements in both built-in and user-defined requirements will be compiled as bigraph patterns for state predicate labelling (②) if the pattern matches the current state then the predicate is true. Step ③ combines bigraph models representing agent programs and CAN semantics [3], and asks BigraphER to explore all possible executions. The output of BigraphER (an explicit transition system with state predicate labels), and built-in and user-defined temporal logic formulae (complied from ④) are then verified by PRISM[1] (⑤). Next, we will go through the features of our tool in detail one by one.

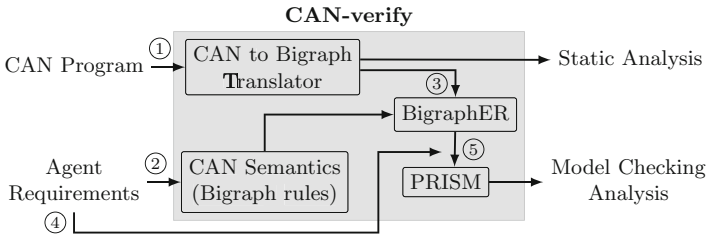


**Fig. 1.** Toolchain overview: ① agent program compilation to bigraphs, ② predicate labelling in bigraph model, ③ (exhaustive) execution of programs, ④ built-in and user-defined belief-based specification formalisation in CTL, ⑤ formal verification.

***Static Analysis of BDI Programs.*** Our tool provides static checks of agent programs including reporting syntax errors, type errors *e.g.* when a plan is used where a belief is expected, and undefined errors *e.g.* when an actions is used but does not exist, or no plan is able to handle a defined event. We also support design aids as warnings, for example, reporting if (customisable) limits are violated such as the mininum/maximum number of plans for an event.

***CAN Interpreter.*** As the bigraph model includes the semantics for the CAN language, given an initial state we can execute any CAN programs. Note: there is no support to actually execute actions, but only to record their outcomes.

[1] Any model checker supporting explicit model import would work. PRISM is chosen as BigraphER natively supports PRISM format, and as ongoing work we aim to allow probabilistic extensions and reasoning on CAN programs [2,5].

**Listing 1.3.** CAN agent for concurrent sensing in UAVs

```
ram_free, storage_free // Initial belief bases
sensing // External events
//Plan library
sensing: true <- dust || photo.
dust: ram_free & storage_free <- collect_dust; analyse; send_back.
photo: ram_free & storage_free <- focus_camera; save_shots; zip_shots.
// Actions description
collect_dust: ram_free <- <{ram_free}, {}>
analyse: true <- <{}, {}>
send_back: storage_free <- <{}, {ram_free,storage_free}>
focus_camera: true <- <{}, {}>
save_shots: storage_free <- <{storage_free}, {}>
zip_shots: ram_free <- <{}, {ram_free, storage_free}>
```

***Model Checking of BDI Programs.*** Model checking is enabled by taking the executable model of the agent program and constructing a labelled transition system of the program's possible executions that allow checking agent requirements against this model. Built-in agent requirements for generic properties include determining if for some/all executions an event finishes with failure or success (these failure or success state are labelled using bigraph predicates [8] automatically). These requirements are translated into branching time temporal logic formulae *e.g.* Computation Tree logic (CTL) [13] for the PRISM model checker. Example requirements are in Listing 1.2. The first 4 requirements are built-in properties that will always be checked by default. The last 2 properties are user-defined requirements, checking on *e.g.* if some desired/avoided beliefs would hold true in all possible agent behaviours. To avoid requiring users to formalise these properties in CTL syntax, properties are instead expressed in natural language. For example, the input of "In all possible executions, eventually the belief `at_venue` holds"[2] equals to the CTL formula `A [ F ("at_venue") ]`, checking eventually the agent of Listing 1.1 arrives at the venue. This translation is performed by the tool as well. Although current support from natural language to formal properties is limited, and an area of future work, we focus on this style of property specification as this is what non-expert users often encounter in practice. Finally, to verify BDI agents starting from different environmental conditions (*i.e.* different initial belief bases), our tool supports verification from parameterised initial belief bases defined in the CAN file. This is useful as the users do not have to run the agent with each initial belief manually, and, importantly, it facilitates a quick comparison of the results from different initial beliefs. Each initial belief set is numbered and the tool automatically runs multiple times to output a result for each initial belief base. For example, we can add another initial belief base *e.g.* `at_shop` in Listing 1.1 to analyse agent behaviours with the initial location at a shop.

## 4 Examples

We give two simple examples to show how our tool improves agent designs and guarantees correct agent behaviours. The aim is to relieve non-expert users from

[2] Parser requires exact natural language wording with user-defined strings as beliefs.

**Listing 1.4.** CAN agent for two-storey building patrol robot.

```
1  //Initial belief bases
2  at_F1, F1_dirty, F1_uninspected, F2_clean, F2_inspected
3  patrol // External events
4  // Plan library
5  patrol: true <- goal(F1_inspected & F2_inspected, check, false).
6  check: at_F1 <- goal(F1_clean, vacuum, false); goal(F1_inspected, inspect
       , false); go_to_F2.
7  check: at_F2 <- goal(F2_clean, vacuum, false); goal(F2_inspected, inspect
       , false); go_to_F1.
8  inspect: at_F1 & F1_uninspected <- inspect_F1.
9  inspect: at_F2 & F2_uninspected <- inspect_F2.
10 vacuum: at_F1 & F1_dirty <- clean_F1.
11 vacuum: at_F2 & F2_dirty <- clean_F2.
12 // Actions description
13 inspect_F1: at_F1 & F1_uninspected <- <{F1_uninspected}, {F1_inspected}>
14 inspect_F2: at_F2 & F2_uninspected <- <{F2_uninspected}, {F2_inspected}>
15 clean_F1: at_F1 & F1_dirty <- <{F1_dirty}, {F1_clean}>
16 clean_F2: at_F2 & F2_dirty <- <{F2_dirty}, {F2_clean}>
17 go_to_F2: at_F1 <- <{at_F1}, {at_F2}>
18 go_to_F1: at_F2 <- <{at_F2}, {at_F1}>
```

the underlying complex translation and verification process by supporting the users' workflow from agent design to agent behaviour analysis.

***Concurrent Sensing.*** We consider a UAV that analyses dust particles and performs aerial photo collection *e.g.* for analysis of post volcanic eruptions. An agent design is in Listing 1.3 from [3]. The external event `sensing` (line 2) initiates the sensing mission, and the relevant plan (line 4) has concurrent (interleaved) tasks for dust monitoring (`dust`) and photo collection (`photo`). Onboard dust sensors require high-speed RAM to collect and analyse the data (`ram_free`), and, when the analysis is complete, results are written to storage (requiring `storage_free`), and sent back to control. Similarly, to collect aerial photos the UAV reserves and focuses the camera array (`focus_camera`), then camera shots are compressed (`zip_shots`), and sent back (where only relevant beliefs are specified in the addition/deletion set of each action (lines 8–13). Static analysis does not identify any issue, however, model checking shows `sensing` is not always completed successfully. A possible explanation is that interleaved concurrency introduces a race condition. To confirm, the agent programmer could replace `dust || photo` (line 4) with `dust; photo` and re-run the tool. The new result shows that the event is now always successful, giving feedback to the agent designer to fix the design through explicit sequencing of actions.

***Multi-storey Building Patrol Robot.*** We consider an autonomous patrol robot that inspects/cleans all floors of a building. For simplicity, the design of two-storey building is given in Listing 1.4. The initial belief base (line 2) gives the location of the robot and the inspection and clean status of each floor. The external event `patrol` (line 3) initiates the patrol mission. The plan (line 5) addressing `patrol` has a true context (always applicable), and a declarative goal whose success condition shows that the event `check` will be pursued until every

floor is inspected. To address the event `check`, plans (lines 6–7) instruct the robot to vacuum and inspect the floor if needed and move to the next higher floor or get to the bottom if the top floor is reached. Plans (lines 8–11) are for actually inspecting and vacuuming. There are no static analysis issues and model checking shows that the event `patrol` is always completed with success, confirming the robot can successfully achieve its task given the initial belief base. To provide an even stronger guarantee of successful cleaning no matter where the robot is initially located, we can parameterise the initial belief base by agent location, *e.g.* adding another initial belief base in which the belief atom `at_F1` changes to `at_F2`, and in each case, the agent exhibits the correct behaviours.

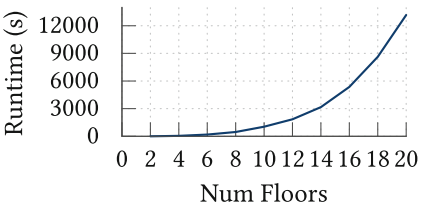


**Fig. 2.** Transition system construction time increases exponentially with floors.

To evaluate scalability, we increase the number of floors. The time to construct the model is in Fig. 2. As expected, there is an exponential increase in time from a couple of seconds to a couple of hours. As our tool is intended to be used at design time we do not believe this to be an issue in practice.

## 5 Discussion and Conclusion

CAN-VERIFY is motivated by the clear need for verification tools that are comprehensible and usable by non-experts. We streamlined the entire underlying process of formal modelling/encoding from BDI agents to Bigraphs, model execution in BigraphER, and model-checking with PRISM. CAN-VERIFY requires only one mandatory input of BDI programs written in the CAN languages and an additional user-defined requirement input file that can be expressed in natural language. The significance of this tool is to meet the growing demands for safe autonomy through formal verification, *e.g.* for early error detection and design improvement, without the costs of applying it *e.g.* formalisation effort.

Current research [4,6] employs the same Bigraph framework to show the use of PRISM and Storm [21] for strategy synthesis and verification of BDI agents and under dynamic environment. We anticipate a small software engineering effort to extend CAN-VERIFY to support both quantitative verification and strategy synthesis. Additionally, the current property specifications mechanism is limited, and we see the task of formalising complex and evolving agent requirements for analysis as a challenging but necessary area. A good starting

point may be to integrate with our tool an existing property elicitation interface such as NASA's Formal Requirements Elicitation Tool (FRET) [18].

**Acknowledgments.** This work is supported by the Engineering and Physical Sciences Research Council, under grants EP/S035362/1, EP/W01081X/1, EP/V026801, and an Amazon Research Award on Automated Reasoning.

## References

1. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Addressing usability in a formal development environment. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12232, pp. 61–76. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_6
2. Archibald, B., Calder, M., Sevegnani, M., Xu, M.: Probabilistic BDI agents: actions, plans, and intentions. In: Calinescu, R., Păsăreanu, C.S. (eds.) SEFM 2021. LNCS, vol. 13085, pp. 262–281. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92124-8_15
3. Archibald, B., Calder, M., Sevegnani, M., Xu, M.: Modelling and verifying BDI agents with bigraphs. Sci. Comput. Program. **215**, 102760 (2022)
4. Archibald, B., Calder, M., Sevegnani, M., Xu, M.: Verifying BDI agents in dynamic environments. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering, pp. 136–141 (2022)
5. Archibald, B., Calder, M., Sevegnani, M., Xu, M.: Quantitative modelling and analysis of BDI agents. Softw. Syst. Model. (2023). https://doi.org/10.1007/s10270-023-01121-5
6. Archibald, B., Calder, M., Sevegnani, M., Xu, M.: Quantitative verification and strategy synthesis for BDI agents. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods. NFM 2023. LNCS, vol. 13903, pp. 241–259. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33170-1_15
7. Bakar, N.A., Selamat, A.: Agent systems verification: systematic literature review and mapping. Appl. Intell. **48**(5), 1251–1274 (2018)
8. Benford, S., Calder, M., Rodden, T., Sevegnani, M.: On lions, impala, and bigraphs: modelling interactions in physical/virtual spaces. ACM Trans. Comput.-Hum. Interact. (TOCHI) **23**(2), 1–56 (2016)
9. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Auton. Agent. Multi-Agent Syst. **12**, 239–256 (2006)
10. Bordini, R.H., et al.: Programming Multi-agent Systems in AgentSpeak using Jason. vol. 8. Wiley, New York (2007)
11. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: Proceedings of IEEE International Conference on Automated Software Engineering, pp. 3–11. IEEE (2000)
12. Cardoso, R.C., et al.: A review of verification and validation for space autonomous systems. Curr. Robot. Rep. **2**(3), 273–283 (2021)
13. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proceedings of Workshop on Logic of Programs, pp. 52–71 (1981)
14. Clavel, M., et al.: Maude manual (version 3.0). SRI International (2020)
15. Dennis, L.A.: Gwendolen semantics: 2017 (2017)

16. Dennis, L.A., et al.: Model checking agent programming languages. Autom. Softw. Eng. **19**(1), 5–63 (2012)
17. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. Electron. Notes Theor. Comput. Sci. **71**, 162–187 (2004)
18. Farrell, M., Luckcuck, M., Sheridan, O., Monahan, R.: FRETting about requirements: formalised requirements for an aircraft engine controller. In: Gervasi, V., Vogelsang, A. (eds.) REFSQ 2022. LNCS, vol. 13216, pp. 96–111. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-98464-9_9
19. Gleirscher, M., van de Pol, J., Woodcock, J.: A manifesto for applicable formal methods. arXiv preprint arXiv:2112.12758 (2021)
20. Hasan, O., Tahar, S.: Formal verification methods. In: Encyclopedia of Information Science and Technology, 3rd Edition, pp. 7162–7170. IGI Global (2015)
21. Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Trans. 1–22 (2021). https://doi.org/10.1007/s10009-021-00633-z
22. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
23. Holzmann, G.J., Lieberman, W.S.: Design and Validation of Computer Protocols, vol. 512. Prentice hall Englewood Cliffs (1991)
24. Jensen, A.B.: Machine-checked verification of cognitive agents. In: Proceedings of the 14th International Conference on Agents and Artificial Intelligence, pp. 245–256 (2022)
25. Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: a survey. ACM Comput. Surv. (CSUR) **52**(5), 1–41 (2019)
26. Milner, R.: The Space and Motion of Communicating Agents. Cambridge University Press, Cambridge (2009)
27. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): : 5. the rules of the game. In: Isabelle/HOL. LNCS, vol. 2283, pp. 67–104. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9_5
28. Pokahr, A., Braubach, L., Jander, K.: The Jadex project: programming model. In: Ganzha, M., Jain, L. (eds.) Multiagent Systems and Applications. Intelligent Systems Reference Library, vol. 45, pp. 21–53. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-33323-1_2
29. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Van de Velde, W., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031845
30. Sardina, S., Padgham, L.: A BDI agent programming language with failure handling, declarative goals, and planning. Auton. Agent. Multi-Agent Syst. **23**(1), 18–70 (2011)
31. Sardina, S., Silva, L.D., Padgham, L.: Hierarchical planning in BDI agent programming languages: a formal approach. In: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1001–1008 (2006)
32. Sevegnani, M., Calder, M.: BigraphER: rewriting and analysis engine for bigraphs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 494–501. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_27
33. Winikoff, M.: Jack$^{TM}$ intelligent agents: an industrial strength platform. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming. MSASSO, vol. 15, pp. 175–193. Springer, Boston, MA (2005). https://doi.org/10.1007/0-387-26350-0_7

34. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative & procedural goals in intelligent agent systems. In: KR, vol. 2002, pp. 470–481 (2002)
35. Xu, M., Rivoalen, T., Archibald, B., Sevegnani, M.: CAN-verify source repository and models, Septemer 2022. https://zenodo.org/record/8282684