Science of Computer Programming 241 (2025) 103233

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original Software Publication

CAN-VERIFY: Automated analysis for BDI agents

Mengwei Xu [a,*], Blair Archibald [b], Michele Sevegnani [b]

[a] School of Computing, Newcastle University, UK
[b] School of Computing Science, University of Glasgow, UK

ARTICLE INFO

Keywords:
BDI agents
Modelling
Verification

ABSTRACT

We present CAN-VERIFY, an automated tool for analysing BDI agents written in the Conceptual Agent Notation (CAN) language. CAN-VERIFY includes support for syntactic error detection before agent execution, agent program interpretation (running agents), and model-checking of agent programs (analysing agents). The model checking supports verifying the correctness of agents against both generic agent requirements, such as if a task is accomplished, and user-defined requirements, such as certain beliefs eventually holding. The latter can be expressed in structured natural language, allowing the tool to be used by agent programmers without formal training in the underlying verification techniques.

1. Motivation

Belief-Desire-Intention agents are a popular model of rational agency. *Beliefs* represent an agent's (possibly incomplete, possibly incorrect) information about itself, other agents, and its environment; *desires* represent the agent's long-term goals; and *intentions* represent the goals that the agent is actively pursuing.

The design of agents is far from trivial due to a mix of declarative specification (a description of the state sought), procedural specification (a set of instructions to perform), failure handling, and inherently interleaved concurrent behaviours (*e.g.* multi-tasking). Crucially, this complexity raises concerns about the safety and trustworthiness of the deployment of these agents.

To tackle these challenges we introduce CAN-VERIFY, an automated verification tool for BDI agents specified using the Conceptual Agent Notation (CAN) language that is a superset of the popular BDI language AgentSpeak [1]. CAN supports sophisticated features such as declarative goals, concurrent processes, and mechanisms for recovering from failures through selecting alternative plans. CAN-VERIFY takes a CAN program and provides several verification capabilities including 1. static analysis of CAN programs, 2. symbolic execution to build the set of all possible executions, and 3. verification through model checking over these executions. We support verification of CAN programs with properties based on both generic agent requirements and optional user-defined requirements in structured natural language. We support verification of agents parameterised by their initial belief base that allows analysis of agent behaviours under different initial environments. This ensures the correct design of the agents before actual deployment under different initial conditions.

* Corresponding author.
E-mail addresses: mengwei.xu@newcastle.ac.uk (M. Xu), blair.archibald@glasgow.ac.uk (B. Archibald), michele.sevegnani@glasgow.ac.uk (M. Sevegnani).

https://doi.org/10.1016/j.scico.2024.103233
Received 28 March 2024; Received in revised form 5 August 2024; Accepted 13 November 2024

Available online 15 November 2024

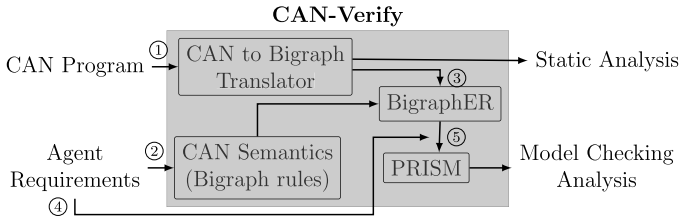

Fig. 1. Toolchain overview: ① agent program compilation to bigraphs, ② predicate labelling in bigraph model, ③ (exhaustive) execution of programs, ④ built-in and user-defined belief-based specification formalisation in CTL, ⑤ formal verification.

```
./CAN-Verify [options] [-p prop.txt] file.can
    -static    Do a static check on the CAN syntax
    -dynamic   Verify the CAN model with BigraphER and PRISM
    -p         Property file for PRISM
    -Ms        Maximum number of states allowed
    -mp        Minimum number of plans required
    -Mp        Maximum number of plans allowed
    -mc        Minimum number of characters required
    -Mc        Maximum number of characters allowed
    -big       Export the CAN model to .big file
    --help     Display this list of options
```

Fig. 2. CAN-VERIFY user options (`-help`).

2. Existing approaches

Existing approaches in automating reasoning for BDI agents are either through model checking or theorem proving. For example, the MCAPL framework [2] implements a BDI language in Java, and verifies it using the Java PathFinder [3] program model checker. This approach verifies the *implementation* of the agents rather than the *semantics* of the language. This can lead to discrepancies. For example, GWENDOLEN [4] (supported by MCAPL) only selects the first applicable plan, while the language semantics describing BDI agents usually allow *any* applicable plan to be chosen. Theorem proving via automated proof assistants such as Isabelle/HOL [5] has been applied to formalise and verify agents defined using the GOAL language [6]. Unlike CAN, GOAL [7] does not allow to select pre-defined plans from a library but instead selects individual actions (or a sequence of actions) in a purely reactive fashion. Crucially, none of the model checking or theorem proving tools are currently fully automated and do not help users with error-prone translation from BDI languages and the specification of complex verification tasks. We provide the first tool that supports automated reasoning about CAN programs without requiring users to have specialist knowledge of verification techniques or formal logic.

3. CAN-VERIFY overview

The theoretical foundation of CAN-VERIFY is to analyse an agent by exploring its possible behaviours via an executable semantics [8]. These semantics are available through an encoding into Milner's Bigraphs [9]—a computational model based on graph-rewriting.[1] We use BigraphER [10] as rewrite engine to compute a labelled transition system and PRISM [11] to analyse it. The detailed software artifacts can be found in Table 1.

3.1. Tool architecture

The architecture of CAN-VERIFY was first presented in detail in [12]. We briefly summarise it here. Fig. 1 shows the logical organisation of our toolchain and how it integrates with external tools. There are five computational steps. Step ① translates input CAN programs into bigraphs expressed in the BigraphER [10] language. During the translation, static checks are performed and errors/warnings reported to users. Step ② translates the static aspects of the requirements as bigraph patterns for state labelling. The intended semantics is that a predicate is assigned to a state if the corresponding pattern is an occurrence. Step ③ combines bigraph models representing agent programs and CAN semantics, and uses BigraphER to produce a labelled transition system (of all possible executions). Step ④ encodes the temporal aspects of the requirements in CTL formulae [13]. Step ⑤ takes the labelled transition system produced by BigraphER and a set of CTL formulae (from ④) as input for PRISM to perform model checking.

For end-users, CAN-VERIFY is implemented as a command line program. Its synopsis is in Fig. 2. The mandatory input is a single CAN program: `file.can`. A minimal example using simple propositional logic is given in Listing 1. The `-static` option performs static analysis of agent programs including reporting type errors *e.g.* when a plan is used where a belief is expected, and undefined errors *e.g.* when an action is used but not defined, or when no plan is specified to handle a defined event.

[1] We support the option `-big` for bigraph-level inspection and analysis.

Listing 1: Example CAN file with parameterised initial belief bases.

```
1  // Initial belief bases
2  1. at_home, car_available
3  2. at_home, car_not_available, no_bus_strike
4  3. at_home, car_not_available, bus_strike
5  // External events
6  travelling
7  //Plan library
8  travelling: at_home&car_available <- drive_to_work.
9  travelling: at_home&car_not_available&no_bus_strike <- bus_to_work.
10 travelling: at_home&car_not_available&bus_strike <- walk_to_work.
11 // Actions description
12 drive_to_work:at_home&car_available<-<{at_home},{at_work}>
13 bus_to_work:at_home&car_not_available&no_bus_strike <-<{at_home},{at_work}>
14 walk_to_work:at_home&car_not_available&bus_strike <-<{at_home},{at_work}>
```

The `-dynamic` option performs model checking. By default this is using built-in agent requirements, *e.g.* for generic properties such as determining if for some/all executions an event finishes with failure or success. Customised properties can be provided through `-p prop.txt` and consist of a list of structured natural language specifications. We focus on a style of property specification that checks if an agent eventually believes some beliefs to be true to allow the user to interrogate the states where the agent may find itself. For example, a user may specify an input of "In all possible executions, eventually the belief `at_work` holds". CAN-VERIFY translates this automatically to the corresponding CTL formula "`A [ F ("at_work") ]`" allowing a model checker (*e.g.* PRISM) to verify if the agent will eventually arrive at the work place. The current implementation of textual property specification is based on simple string matching, so requires exact wording and structure with some fixed free variable locations, *e.g.* holes for the beliefs the user can specify. This captures common properties, *i.e.* checking a belief holds, but does not capture all possible properties (that could be specified in CTL; and experienced users can manually provide a CTL property). Translating from generic natural language to CTL formulas is outwith the scope of this work. A potential solution would be to integrate with our tool an existing property elicitation interface such as NASA's Formal Requirements Elicitation Tool (FRET) [14].

CAN-VERIFY allows the verification of BDI agents under a family of initial conditions. This is supported by allowing multiple initial belief bases to be defined in the CAN file. For example, the program in Listing 1 specifies three initial belief bases corresponding to three different situations for an agent to start with on getting to the work place from home. This eliminates the need for manual runs for each set of belief base, and enables comparison of outcomes across different initial conditions. The next section of code in Listing 1 is marked by comment `External events` in line 5. It contains a list of events for the agent to address; in this case, it is to plan a trip (*i.e.* `travelling`). Section `Plan library` starting in line 7 gives the plans to address the events under different situations. For example, the first plan in line 8, which can address the event `travelling`, says that if the car is available, then the agent can drive to work from home. Finally, in section `Actions description`, actions are defined in terms of pre-conditions and belief base updates. For example, action `drive_to_work` in line 12 says that it is applicable if it is at home and a car is available. Executing this action will result in deleting belief `at_home` and adding `at_work` into the belief base.

4. Planned future development

New theoretical breakthroughs in executable semantics will be integrated with CAN-VERIFY. For example, we provide a probabilistic semantics of CAN allowing quantitative verification, *e.g.* probabilistic plan selection policies and probabilistic action outcomes [15,16]. We can also perform strategy synthesis, *e.g.* what plan to select next, utilising Action Bigraphs [17] which allow non-determinism. We plan to extend agents with uncertain beliefs. In all cases, as we already build on bigraphs, we anticipate a small software engineering effort to extend CAN-VERIFY to incorporate the new encodings.

Metadata

Table 1
Code metadata.

Nr.	Code metadata description	
C1	Current code version	v1
C2	Permanent link to code/repository used for this code version	https://github.com/Mengwei-Xu/CAN-Verify-SCP-Tool
C3	Permanent link to Reproducible Capsule	https://github.com/Mengwei-Xu/CAN-Verify-SCP-Tool
C4	Legal Code License	Simplified BSD License
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	OCaml
C7	Compilation requirements, operating environments and dependencies	BigraphER [10] and PRISM [11]
C8	If available, link to developer documentation/manual	https://github.com/Mengwei-Xu/CAN-Verify-SCP-Tool
C9	Support email for questions	mengwei.xu@newcastle.ac.uk

CRediT authorship contribution statement

Mengwei Xu: Writing – original draft. **Blair Archibald:** Writing – review & editing. **Michele Sevegnani:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is supported the Newcastle University (UK), EPSRC grants EP/S035362/1, and an Amazon Research Award on Automated Reasoning.

References

[1] A.S. Rao, AgentSpeak (L): BDI agents speak out in a logical computable language, in: European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Springer, 1996, pp. 42–55.
[2] L.A. Dennis, et al., Model checking agent programming languages, Autom. Softw. Eng. 19 (1) (2012) 5–63.
[3] G. Brat, K. Havelund, S. Park, W. Visser, Model checking programs, in: Proceedings of IEEE International Conference on Automated Software Engineering, IEEE, 2000, pp. 3–11.
[4] L.A. Dennis, Gwendolen semantics: 2017, 2017.
[5] T. Nipkow, M. Wenzel, L.C. Paulson, Isabelle/HOL: a Proof Assistant for Higher-Order Logic, Springer, 2002.
[6] A.B. Jensen, Machine-checked verification of cognitive agents, in: Proceedings of the 14th International Conference on Agents and Artificial Intelligence, 2022, pp. 245–256.
[7] K.V. Hindriks, F.S. De Boer, W. Van Der Hoek, J.-J.C. Meyer, Agent programming with declarative goals, in: Intelligent Agents VII Agent Theories Architectures and Languages: 7th International Workshop, Proceedings, ATAL 2000 Boston, MA, USA, July 7–9, 2000, vol. 7, Springer, 2001, pp. 228–243.
[8] B. Archibald, M. Calder, M. Sevegnani, M. Xu, Modelling and verifying BDI agents with bigraphs, Sci. Comput. Program. 215 (2022) 102760.
[9] R. Milner, The Space and Motion of Communicating Agents, Cambridge University Press, 2009.
[10] M. Sevegnani, M. Calder, BigraphER: rewriting and analysis engine for bigraphs, in: Proceedings of International Conference on Computer Aided Verification, Springer, 2016, pp. 494–501.
[11] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: verification of probabilistic real-time systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proc. 23rd International Conference on Computer Aided Verification (CAV'11), in: LNCS, vol. 6806, Springer, 2011, pp. 585–591.
[12] M. Xu, T. Rivoalen, B. Archibald, M. Sevegnani, CAN-Verify: a verification tool for BDI agents, in: International Conference on Integrated Formal Methods, Springer, 2023, pp. 364–373.
[13] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: D. Kozen (Ed.), Logics of Programs, Workshop, Yorktown Heights, New York, USA, in: Lecture Notes in Computer Science, vol. 131, May 1981, pp. 52–71.
[14] M. Farrell, M. Luckcuck, O. Sheridan, R. Monahan, Fretting about requirements: formalised requirements for an aircraft engine controller, in: Requirements Engineering: Foundation for Software Quality: 28th International Working Conference, Proceedings, REFSQ 2022, Birmingham, UK, March 21–24, 2022, Springer, 2022, pp. 96–111.
[15] B. Archibald, M. Calder, M. Sevegnani, M. Xu, Probabilistic BDI agents: actions, plans, and intentions, in: Proceedings of Software Engineering and Formal Methods, Springer International Publishing, 2021, pp. 262–281.
[16] B. Archibald, M. Calder, M. Sevegnani, M. Xu, Quantitative modelling and analysis of BDI agents, Softw. Syst. Model. (2023).
[17] B. Archibald, M. Calder, M. Sevegnani, Probabilistic bigraphs, Form. Asp. Comput. 34 (2) (2022) 1–27.