Science of Computer Programming 242 (2025) 103254

Contents lists available at ScienceDirect

Science of Computer Programming

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

# Modelling and verifying BDI agents under uncertainty

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

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

A R T I C L E   I N F O

*Keywords:*
BDI agents
Uncertain beliefs
Modelling
Verification
Synthesis

A B S T R A C T

Belief-Desire-Intention (BDI) agents feature uncertain beliefs (e.g. sensor noise), probabilistic action outcomes (e.g. attempting and action and failing), and non-deterministic choices (e.g. what plan to execute next). To be safely applied in real-world scenarios we need reason about such agents, for example, we need probabilities of mission success and the *strategies* used to maximise this. Most agents do not currently consider uncertain beliefs, instead a belief either holds or does not. We show how to use epistemic states to model uncertain beliefs, and define a Markov Decision Process for the semantics of the Conceptual Agent Notation (CAN) agent language allowing support for uncertain beliefs, non-deterministic event, plan, and intention selection, and probabilistic action outcomes. The model is executable using an automated tool—CAN-VERIFY—that supports error checking, agent simulation, and exhaustive exploration via an encoding to Bigraphs that produces transition systems for probabilistic model checkers such as PRISM. These model checkers allow reasoning over quantitative properties and strategy synthesis. Using the example of an autonomous submarine and drone surveillance together with scalability experiments, we demonstrate our approach supports uncertain belief modelling, quantitative model checking, and strategy synthesis in practice.

## 1. Introduction

The Belief-Desire-Intention (BDI) [1] architecture is a popular and well-studied rational agent framework and forms the basis of, among others, AgentSpeak [2], An Abstract Agents Programming Language (3APL) [3], A Practical Agent Programming Language (2APL) [4], Jason [5], and Conceptual Agent Notation (CAN) [6]. In a BDI agent, the *(B)eliefs* represent what the agent knows, the *(D)esires* what the agent wants to bring about, and the *(I)ntentions* those desires the agent has chosen to act upon.

In BDI languages, desires and intentions are represented implicitly by defining a plan library where the plans are written by programmers in a modular fashion. Plans describe how, and under what conditions (based on beliefs), an agent can react to an event (a desire). The set of intentions are those plans that are currently being executed. A desirable feature of agent-based systems is that they are reactive [7]: an agent can respond to new events even while already dealing with existing events. To allow this, agents pursue multiple events and execute intentions in an interleaved manner. This requires a decision-making process: which event to handle first (event selection) and which intention to progress next (intention selection). When handling events, we must also decide which plan to select from a set of possible plans (plan selection).

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

https://doi.org/10.1016/j.scico.2024.103254
Received 21 June 2024; Received in revised form 15 November 2024; Accepted 26 November 2024

Available online 3 December 2024

The deployment of BDI-based systems raises concerns of trustworthiness. In practice, the *beliefs of an agent are uncertain*, for example due to sensor noise, incomplete information, or generation by a probabilistic algorithm, e.g. machine learning. In a multi-agent environment, new information from other agents may not cancel out existing beliefs but instead strengthen or weaken them. *Erroneous plans* can cause incorrect behaviour. Even with a correct plan library, careless decisions for interleaving intention progression can result in failures/conflicts, e.g. the progressing one intention can make it impossible to progress another (deadlock). This negative tension between modularised plan design and interleaved execution is difficult to identify using traditional non-exhaustive testing approaches as there is no guarantee we see all interleavings. *Action outcomes are inherently probabilistic* due to imprecise actuation. As a result, there is a growing need for formal techniques that can handle quantitative properties of agent-based systems, in particular, under uncertainty [8]. Given the number of non-deterministic decisions faced by an agent, we may want to synthesise a strategy to determine ahead-of-time the decisions an agent should make, e.g. to avoid the worst-case execution.

To illustrate the challenges, we use a robotic submarine example. The goal of the robotic submarine is to inspect a pipeline located on a seabed. The robotic submarine has the (non-deterministic) choice to operate at different depths: low, medium, or high. Low depths allow a wider field of vision (and so faster mission completion). But, at low depths (far away from the seabed), the accuracy of the vision algorithm decreases resulting in uncertain beliefs. Higher depths increase vision accuracy but also increase the probability of thruster failure, e.g. due to seaweed wrapping around the thrusters. It is important to compare the trade-offs between vision accuracy and thruster failure, and in general we need to model and quantify agent behaviour when there are a range of non-deterministic choices, uncertain beliefs, and probabilistic action outcomes. For example, we may want to guarantee the submarine completes the mission with a probability above a threshold.

Verifying BDI agent behaviours through model checking and theorem proving has been well explored. For example, using the Java PathFinder model-checker [9] and the Isabelle/HOL proof assistant [10]. Unfortunately, they do not adequately represent agent behaviours in cyber-physical robotics systems (e.g. surveyed in [11]), e.g. supporting modelling imprecise actuators. To reason with the quantitative behaviours of BDI agents, the authors of [12] investigate the probabilistic semantics and resulting verification of BDI agents with imprecise actuators by resolving non-determinism in various selections through manually specified strategies (fixed orders, round-robin fashion, or probabilistic distribution). However, these hand-crafted strategies may not be optimal. Determining effective strategies is complex and often requires advanced planning algorithms [13,14].

We show how to enable quantitative verification and strategy synthesis [15,16] for BDI agents. This allows us to determine the probability an agent successfully completes a mission under uncertain beliefs (and probabilistic actions), and derive a suitable strategy for resolving the non-deterministic choices in intention/event/plan selection that maximises success. We focus on the CAN language [17,18], a high-level agent programming language that captures the essence of BDI concepts without describing implementation details. As a superset of most well-known AgentSpeak [2], CAN includes advanced BDI agent behaviours such as reasoning with *declarative goals* and *failure recovery*. Although we focus on CAN, the language features are similar to those of other mainstream BDI languages and the same modelling and verification techniques would apply to other BDI programming languages.

We build on an executable *non-deterministic* and *probabilistic* semantics of CAN [12], based on Milner's Bigraphs [19]. Specifically, we use action bigraphs [20] to model plan/intention selection as non-deterministic choices and assign probabilities to system transitions (graph rewrites). This provides a model of CAN based on a Markov decision process (MDP) [21] that we denote as $\text{CAN}^m$. The MDP formalisation of agent behaviours enables us to model certain unknown aspects of a system's behaviour, e.g. the scheduling between intentions executing in parallel and represent uncertainty arising from, for example, action execution. However, it omits the important part of uncertainty arising in the agent's beliefs. To address this shortcoming, we present in this paper an extension of $\text{CAN}^m$, denoted as $\text{CAN}^m_+$, where the uncertain beliefs of an agent are modelled by epistemic states as described in [22,23]. Epistemic states represent the degree of certainty in the agent's beliefs. For analysis we use BigraphER [24]—a framework for manipulating bigraphs—that executes a rewrite system over a given initial state (initial agent program setup). The set of all states and transitions from BigraphER is the underlying MDP which can be analysed using existing probabilistic model checkers, e.g. PRISM [25]. Model checkers allow checking of probabilistic/reward-based properties based on (probabilistic) temporal logics, and can perform strategy synthesis (to determine which actions to take in the face of non-determinism).

To widen the access of our modelling and verification techniques, we provide an automated tool, CAN-VERIFY, that automates the translation from agent specification to bigraphs. CAN-VERIFY can also provide (probabilistic) verification against both generic agent requirements, such as determining the probability a task is successful, and user-defined requirements, such as determining the probability a certain belief is true at some point in the future. This is done by automatically calling PRISM on the MDP generated by BigraphER.

Parts of this study and preliminary results were presented in [26]. We make the following additional research contributions:

- We extend the operational semantics of the CAN language [22,23] to support uncertain beliefs by utilising epistemic states [22]. We call this new language $\text{CAN}_+$;
- Via a translation to (action) bigraphs, we provide an executable model of $\text{CAN}_+$ that we call $\text{CAN}^m_+$. Using BigraphER, we can construct the underlying MDP (that defines which actions are allowed in which states) and this can be passed to PRISM for quantitative verification and strategy synthesis;
- We give an extended version of the CAN-VERIFY tool [27] that supports agents with uncertain beliefs;
- We show how these new semantics support quantitative analysis using a robotic submarine example and scalability experiments in a drone example;

The paper is organised as follows: Section 2 introduces BDI agents, the CAN language, and MDPs; in Section 3, we show how epistemic states can be used to model uncertain beliefs and give an extended semantics for CAN, i.e. $\text{CAN}_+$; in Section 4, we show how $\text{CAN}_+$ can be formalised as an MDP. In Section 5, we show how to encode the MDP model in Bigraphs and how to perform probabilistic verification and strategy synthesis for a robotic submarine. We also present our extension of CAN-VERIFY for probabilistic verification of agents' properties. In Section 7, we reflect on our choices of epistemic states to model uncertain beliefs and bigraphs as an encoding language. Section 8 discusses related work, Section 9 gives plans of for future extensions, and we conclude in Section 10.

## 2. Preliminaries

We give an overview of BDI agents using the Conceptual Agent Notation (CAN) language, and Markov Decision Processes (MDPs).

### 2.1. BDI agents

A BDI agent has an explicit representation of beliefs, desires, and intentions. The beliefs correspond to what the agent believes about the environment, while the desires are a set of *external* events that the agent can respond to. To respond to those events, the agent selects an appropriate plan (given its beliefs) from the pre-defined plan library and commits to the selected plan by turning it into a new intention.

#### 2.1.1. Syntax

CAN is a superset of AgentSpeak [2] featuring the same core operational semantics, along with several additional appealing features: declarative goals, concurrency, and failure handling.

A CAN agent consists of a belief base $\mathcal{B}$ and a plan library $\Pi$. The *belief base* $\mathcal{B}$ encodes the current beliefs (usually as atoms, or first order predicates) and has operators to check whether a belief formula $\varphi$ follows from the belief base (i.e. $\mathcal{B} \vDash \varphi$), and to revise the belief base. The standard way to perform for belief revision is to add a belief atom $b$ to a belief base $\mathcal{B}$ (i.e. $\mathcal{B} \cup \{b\}$), and to delete a belief atom from a belief base (i.e. $\mathcal{B} \setminus \{b\}$).

A *plan library* $\Pi$ contains the operational procedures of an agent and is a finite collection of plans of the form $Pl = e : \varphi \leftarrow P$ with $Pl$ the plan identifier, $e$ the triggering event, $\varphi$ the context condition, and $P$ the plan-body. The triggering event $e$ specifies why the plan is triggered, the context condition $\varphi$ determines *when* the plan-body $P$ is able to handle the event. For convenience, we call the set of events from the external environment the external event set, denoted $E^e$. Finally, the remaining events (which occur as a part of the plan-body) are either sub-events or internal events.

By convention (e.g. in [5]), the set of plan-bodies $P$ in a plan $Pl = e : \varphi \leftarrow P$ may be referred to as the *program* or *agent program* and has the following syntax:

$$P ::= act \mid ?\varphi \mid +b \mid -b \mid e \mid P_1;P_2 \mid P_1 \parallel P_2 \mid goal(\varphi_s, P, \varphi_f)$$

with $act$ an action, $?\varphi$ a test for $\varphi$ entailment in the belief base, $+b$ and $-b$ represent belief addition and deletion, and $e$ is a sub-event (i.e. internal event). To execute a sub-event, a plan (corresponding to that event) is selected and the plan-body added in place of the event. In this way we allow plans to be nested (similar to sub-routine calls in other languages). Actions $act$ take the form $act = \psi \leftarrow \langle \phi^+, \phi^- \rangle$, where $\psi$ is the pre-condition, and $\phi^+$ and $\phi^-$ are the addition and deletion sets (resp.) of belief atoms, i.e. a belief base $\mathcal{B}$ is revised with addition and deletion sets $\phi^+$ and $\phi^-$ to be $(\mathcal{B} \setminus \phi^-) \cup \phi^+$ when the action executes. In addition, there are composite programs $P_1;P_2$ for sequence and $P_1 \parallel P_2$ for interleaved concurrency. Finally, a declarative goal program $goal(\varphi_s, P, \varphi_f)$ expresses that the declarative goal $\varphi_s$ should be achieved through program $P$, failing if $\varphi_f$ becomes true, and retrying as long as neither $\varphi_s$ nor $\varphi_f$ is true (see in [18] for details). Additionally, there are auxiliary program forms that are used internally when assigning semantics to programs, namely $nil$, the empty program, and $P_1 \rhd P_2$ that executes $P_2$ if the case that $P_1$ fails. When a plan $Pl = e : \varphi \leftarrow P$ is selected to respond to an event, its plan-body $P$ is adopted as an intention in the intention base $\Gamma$ (a.k.a. the partially executed plan-body). Finally, we assume a plan library does not have recursive plans (thus avoiding potential infinite state space).

#### 2.1.2. Semantics

CAN semantics is specified by two types of transitions. The first, denoted $\rightarrow$, specifies *intention-level* evolution on configurations $\langle \mathcal{B}, P \rangle$ where $\mathcal{B}$ is the belief base, and $P$ the plan-body currently being executed. The second type, denoted $\Rightarrow$, specifies *agent-level* evolution over $\langle E^e, \mathcal{B}, \Gamma \rangle$, detailing how to execute a complete agent where $E^e$ is the set of pending external events to address (a.k.a. desires), $\mathcal{B}$ the belief base, and $\Gamma$ a set of partially executed plan-bodies (intentions).

Fig. 1 summarises rules for evolving any single intention. For example, the rule $act$ handles the execution of an action, when the pre-condition $\psi$ is met, resulting in a belief state update. Rule $event$ replaces an event with the set of relevant plans, while rule $select$ chooses an applicable plan from a set of relevant plans while retaining un-selected plans as backups. With these backup plans, the rules for failure recovery $\rhd_;$, $\rhd_\top$, and $\rhd_\bot$ enable new plans to be selected if the current plan fails (e.g. due to environment changes). Rules ; and $;_\top$ allow executing plan-bodies in sequence, while rules $\parallel_1$, $\parallel_2$, and $\parallel_\top$ specify how to execute (interleaved) concurrent programs. Rules $G_s$ and $G_f$ deal with declarative goals when either the success condition $\varphi_s$ or the failure condition $\varphi_f$ become true. Rule $G_{init}$ initialises persistence by setting the program in the declarative goal to be $P \rhd P$, i.e. if $P$ fails try $P$ again, and rule $G_;$ takes care of performing a single step on an already initialised program. Finally, the derivation rule $G_\rhd$ re-starts the original program if the current program has finished or got blocked (when neither $\varphi_s$ nor $\varphi_f$ is true).

$$\frac{act : \psi \leftarrow \langle \phi^-, \phi^+ \rangle \quad \mathcal{B} \vDash \psi}{\langle \mathcal{B}, act \rangle \rightarrow \langle (\mathcal{B} \setminus \phi^- \cup \phi^+), nil \rangle}\, act \quad \frac{\Delta = \{\varphi : P \mid (e' = \varphi \leftarrow P) \in \Pi \wedge e' = e\}}{\langle \mathcal{B}, e \rangle \rightarrow \langle \mathcal{B}, e : (\!| \Delta |\!) \rangle}\, event \quad \frac{\varphi : P \in \Delta \quad \mathcal{B} \vDash \varphi}{\langle \mathcal{B}, e : (\!| \Delta |\!) \rangle \rightarrow \langle \mathcal{B}, P \rhd e : (\!| \Delta \setminus \{\varphi : P\} |\!) \rangle}\, select$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, P_1 \rhd P_2 \rangle \rightarrow \langle \mathcal{B}', P_1' \rhd P_2 \rangle)}\rhd_; \quad \frac{}{\langle \mathcal{B}, (nil \rhd P_2) \rangle \rightarrow \langle \mathcal{B}', nil \rangle}\rhd_\top \quad \frac{P_1 \neq nil \ \langle \mathcal{B}, P_1 \rangle \nrightarrow \ \langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}{\langle \mathcal{B}, P_1 \rhd P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle} \ \rhd_\bot \quad \frac{\langle \mathcal{B}, P \rangle \rightarrow \langle \mathcal{B}', P' \rangle}{\langle \mathcal{B}, (nil; P) \rangle \rightarrow \langle \mathcal{B}', P' \rangle} ;_\top$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, (P_1 ; P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1' ; P_2) \rangle} ; \quad \frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1' \| P_2) \rangle} \|_1 \quad \frac{\langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1 \| P_2') \rangle} \|_2 \quad \frac{}{\langle \mathcal{B}, (nil \| nil) \rangle \rightarrow \langle \mathcal{B}, nil \rangle} \|_\top$$

$$\frac{\mathcal{B} \vDash \varphi_s}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, nil \rangle} G_s \quad \frac{\mathcal{B} \vDash \varphi_f}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, ?false \rangle} G_f \quad \frac{P \neq P_1 \rhd P_2 \quad \mathcal{B} \nvDash \varphi_s \quad \mathcal{B} \nvDash \varphi_f}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, goal(\varphi_s, P \rhd P, \varphi_f) \rangle} G_{init}$$

$$\frac{\mathcal{B} \nvDash \varphi_s \quad \mathcal{B} \nvDash \varphi_f \quad \langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f) \rangle \rightarrow \langle \mathcal{B}', goal(\varphi_s, P_1' \rhd P_2, \varphi_f) \rangle} G_; \quad \frac{\mathcal{B} \nvDash \varphi_s \quad \mathcal{B} \nvDash \varphi_f \quad \langle \mathcal{B}, P_1 \rangle \nrightarrow}{\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, goal(\varphi_s, P_2 \rhd P_2, \varphi_f) \rangle} G_\rhd$$

**Fig. 1.** Intention-level CAN semantics.

$$\frac{e \in E^e}{\langle E^e, \mathcal{B}, \Gamma \rangle \Rightarrow \langle E^e \setminus \{e\}, \mathcal{B}, \Gamma \cup \{e\} \rangle} A_{event} \quad \frac{P \in \Gamma \quad \langle \mathcal{B}, P \rangle \rightarrow \langle \mathcal{B}', P' \rangle}{\langle E^e, \mathcal{B}, \Gamma \rangle \Rightarrow \langle E^e, \mathcal{B}', (\Gamma \setminus \{P\}) \cup \{P'\} \rangle} A_{step} \quad \frac{P \in \Gamma \quad \langle \mathcal{B}, P \rangle \nrightarrow}{\langle E^e, \mathcal{B}, \Gamma \rangle \Rightarrow \langle E^e, \mathcal{B}, \Gamma \setminus \{P\} \rangle} A_{update}$$

**Fig. 2.** Agent-level CAN semantics.

The agent-level semantics are given in Fig. 2. The rule $A_{event}$ handles external events by adopting them as intentions. Rule $A_{step}$ selects an intention from the intention base, and evolves a single step w.r.t. the intention-level transition, while $A_{update}$ discards any unprogressable intentions (either already succeeded, or failed).

## 2.2. Markov decision processes

Markov decision processes (MDPs) are a widely used formalism for modelling systems that exhibit both probabilistic actions and nondeterministic choices. An MDP [21] is a tuple $\mathcal{M} = (S, \bar{s}, \alpha, \delta)$ where $S$ is a set of states, $\bar{s}$ an initial state, $\alpha$ a set of actions (atomic labels), and $\delta : S \times \alpha \rightarrow Dist(S)$ a (partial) probabilistic transition function where $Dist(S)$ is the set of the probability distribution over states $S$. Each state $s$ of an MDP $\mathcal{M}$ has a (possibly empty) set of *enabled* actions $A(s) \overset{\text{def}}{=} \{a \in \alpha \mid \delta(s, a) \text{ is defined}\}$. When action $a \in A(s)$ is taken in state $s$, the next state is determined probabilistically according to the distribution $\delta(s, a)$, i.e. the probability that a transition to state $s'$ occurs is $\delta(s, a)(s')$. An MDP may have an action reward structure i.e. a function of the form $R : S \times \alpha \rightarrow \mathbb{R}_{\geq 0}$ that increments a counter when an action is taken. An *adversary* (also known as a strategy or policy) resolves non-determinism by determining a single action choice per state, and optimal adversaries are those that e.g. minimise the probability some property holds. This can be used to ensure, for example, the chance of system failure events is minimised.

## 3. Epistemic beliefs in BDI agents

Traditionally, the belief base of a BDI agent is a set of belief atoms that are believed either true or false by an agent. However, in a realistic setting the beliefs of an agent are uncertain. This might be because of technical imprecision, such as sensor noise, or where the information is coming from e.g. if it was produced by a machine learning algorithm. Likewise, when gathering information from other agents, to avoid security issues, information should not be treated as fact, but only as evidence *towards* some belief atom. Given this, a binary model for belief atoms is too weak in practice, and instead, we need a belief model with a level of confidence in certain belief atoms.

To allow degrees of belief, we use results from the field of belief revision. In particular, the idea of an *epistemic state* applied to agents. Epistemic states allow us to associate a *plausibility* to a particular set of belief atoms. A full description of epistemic states is in [22,23], and we reuse these notations, which are publicly available, as the preparation to our modelling later on.

For epistemic states, degrees of a set of beliefs (likelihoods) are measured numerically i.e. as a function $\omega$ mapping a set of beliefs to $\mathbb{Z} \cup \{-\infty, +\infty\}$, where integers closer to $+\infty$ imply more confidence in the set of beliefs, i.e. $+\infty$ means full confidence, and likewise for less confidence as we tend towards $-\infty$.

## 3.1. Modelling uncertain beliefs

We start from a finite set of belief atoms $\mathcal{A}t$, and $Lit = \{a \mid a \in \mathcal{A}t\} \cup \{\neg a \mid a \in \mathcal{A}t\}$ be the set of literals (truth assignments) constructed from $\mathcal{A}t$. For a literal $l \in Lit$, we use $l^*$ to denote the underlying atom such that $l^* = a$ when $l = a$ or $l = \neg a$. The main idea is, instead of assigning a plausibility to a set of belief atoms, we track for each belief atom evidence *for* and *against* that belief atom being true. If required, we can recover the plausibility of the entire belief base through a linear combination[1] of the plausibility of each belief atom.

Epistemic states use $\pm\infty$ to denote absolute confidence/distrust in a particular atom, i.e. truth and falsehood. In practice we do not want atoms to be considered absolutely true, so we exclude the possibility of $\pm\infty$ in our model.[2]

1 We assume that belief atoms are independent so one belief holding does not affect the likelihood of another holding.

2 This could be re-added if absolute truth was required, with careful consideration on for belief revision as $+$ is not defined over $\pm\infty$.

**Definition 1.** Let $\mathcal{A}t$ be a finite set of belief atoms. A compact epistemic state $\mathcal{W}$ is a mapping $\mathcal{W} : \mathcal{A}t \mapsto (\mathbb{Z}, \mathbb{Z})$. We denote the mapping of a single atom $a$ as $\mathcal{W}(a) = (\mu_+, \mu_-)$ where $\mu_+$ is evidence for the atom being true and $\mu_-$ evidence for the atom being false.

**Definition 2.** For a compact epistemic state $\mathcal{W}$ and atom $a \in \mathcal{A}t$, the likelihood belief $a$ is true in this state (i.e. $a$, not $\neg a$) is denoted as $\mathcal{W} \vDash a$ and holds when $\mathcal{W}(a) = (\mu_+, \mu_-)$ and $\mu_+ - \mu_- > 0$. That is, there is more evidence it holds than it does not. When $\mu_+ - \mu_- < 0$ then $\mathcal{W} \vDash \neg a$. For $\mu_+ - \mu_- = 0$ we do not have any clear evidence so do not hold either belief.

Throughout an agents lifetime, we need to revise the beliefs based on new information/evidence (e.g. through sensors) as follows:

**Definition 3.** A new piece of information concerning a belief literal $l \in Lit$ for atom $a$ with degree of belief $m$ is a compact epistemic state $\mathcal{W}_{new}$ such that

- $\mathcal{W}_{\text{new}}(a) = (m, 0)$ if $l = a$
- $\mathcal{W}_{\text{new}}(a) = (0, m)$ if $l = \neg a$

For convenience, we denote the new compact epistemic state for a literal $l$ and degree of belief as $(l, m)$, e.g. $(a, 2)$ etc.
Finally, we define the revision of a compact epistemic state $\mathcal{W}$ by another epistemic state $\mathcal{W}'$, denoted as $\mathcal{W} \circ \mathcal{W}'$.

**Definition 4.** For two compact epistemic states $\mathcal{W}$, $\mathcal{W}'$ where $\mathcal{W}(a) = (\mu_+, \mu_-)$ and $\mathcal{W}'(a) = (\mu'_+, \mu'_-)$, the revised (combined) epistemic state $\mathcal{W}'' = \mathcal{W} \circ \mathcal{W}'$ such that (for all atoms) $\mathcal{W}''(a) = (\mu_+ + \mu'_+, \mu_- + \mu'_-)$. Notice that the commutativity of $+$ means it does not matter which order we apply revisions in.

For an input compact epistemic state $(l, m)$ (which is usually what we revise over), we denote $\mathcal{W}' = \mathcal{W} \circ (l, m)$ where

$$\mathcal{W}(a) = \begin{cases} (\mu_+ + m, \mu_-) & \text{if } l = a \\ (\mu_+, \mu_- + m) & \text{if } l = \neg a \\ \mathcal{W}(a) & \text{otherwise} \end{cases}$$

**Example 1.** Let $\mathcal{A}t$ be $\{a, b, c\}$ with an initial compact epistemic state $\mathcal{W}(a) = \mathcal{W}(b) = \mathcal{W}(c) = (0, 0)$. Applying a sequence of compact epistemic inputs $(\neg c, 2), (a, 4), (b, -3), (\neg a, 3)$, we obtain a new epistemic state $\mathcal{W}'$ where $\mathcal{W}'(a) = (4, 3)$, $\mathcal{W}'(b) = (-3, 0)$, and $\mathcal{W}'(c) = (0, 2)$. $\mathcal{W}' \vDash a$ as $\mathcal{W}' = (4, 3)$ and $\mu_+ - \mu_- = 4 - 3 > 0$.

### *3.2. Semantics for epistemic beliefs in CAN*

BDI agents are already structured to allow any logic that allows belief updates and an entailment operator. We utilise this to use the compact epistemic state $\mathcal{W}$ to take the role of a belief base $\mathcal{B}$ in CAN, giving us the updated semantics CAN$_+$. From now on, we will focus on CAN$_+$ (that supports uncertainty beliefs) to avoid confusion. Only minor updates are required in belief entailment and belief revision rules. For example, the new semantics for action execution allows *weighted* belief updates in the original CAN:

$$\frac{act : \psi \leftarrow (l, m) \quad \mathcal{W} \vDash \psi}{\langle \mathcal{W}, act \rangle \to \langle \mathcal{W} \circ (l, m), nil \rangle} \; act$$

This states the effect of an action is an input compact epistemic state $(l, m)$ in $act : \psi \leftarrow (l, m)$. The intention-level configuration of CAN changes from $\langle \mathcal{B}, act \rangle$ to $\langle \mathcal{W}, act \rangle$ to show we are using epistemic beliefs. Finally, the effect of an action revises the compact epistemic state $\mathcal{W}$ with the input compact epistemic state $(l, m)$, i.e. $\mathcal{W} \circ (l, m)$. The rest of semantics rules can be similarly adjusted. Before we close this section, we also notice that the action execution in the original CAN is non-probabilistic (so is the case with CAN$_+$) and we will extend it to probabilistic actions in the next section.

## 4. An MDP model of CAN$_+$ semantics

MDPs model systems with nondeterministic and probabilistic behaviour. To use an MDP with the CAN$_+$, we associate CAN$_+$ rules with MDP actions and CAN$_+$ states to MDP states. We refer to the MDP model of CAN$_+$ as CAN$_+^m$.

States in CAN$_+^m$ are given by the agent-level configuration $\langle E^e, \mathcal{W}, \Gamma \rangle$ of CAN$_+$. The state space is $S \subseteq 2^{E^e} \times 2^{\mathcal{W}} \times 2^{\Gamma}$ where the exact subset of states is determined by the specific program we are modelling and we abuse the notation $2^{\mathcal{W}}$ to stand for all possible instances of compact epistemic states.[3] An initial state of a CAN$_+^m$ is $\bar{s} = \langle E_0^e, \mathcal{W}_0, \Gamma_0 \rangle$. For example, it can have the form $E_0^e = \{e_1, \cdots, e_j\}$ (a set of tasks), $\mathcal{W}_0$ such that $\mathcal{W}(b_i) = (0, 0)$ (an initial set of beliefs, e.g. about the environment), $\Gamma_0 = \emptyset$ (no intentions yet), and $i, j \in \mathbb{N}^+$.

The original CAN semantics are defined using operational semantics with transitions over configurations $C \to C'$ (so is CAN$_+$) that denote a single execution step between configuration $C$ and $C'$ (see Section 2.1.2). As we reason with probabilistic action outcomes

[3] We determine this by symbolically executing the program as we convert to an MDP, and in practice we consider a finite subset of $2^{\mathcal{W}}$.

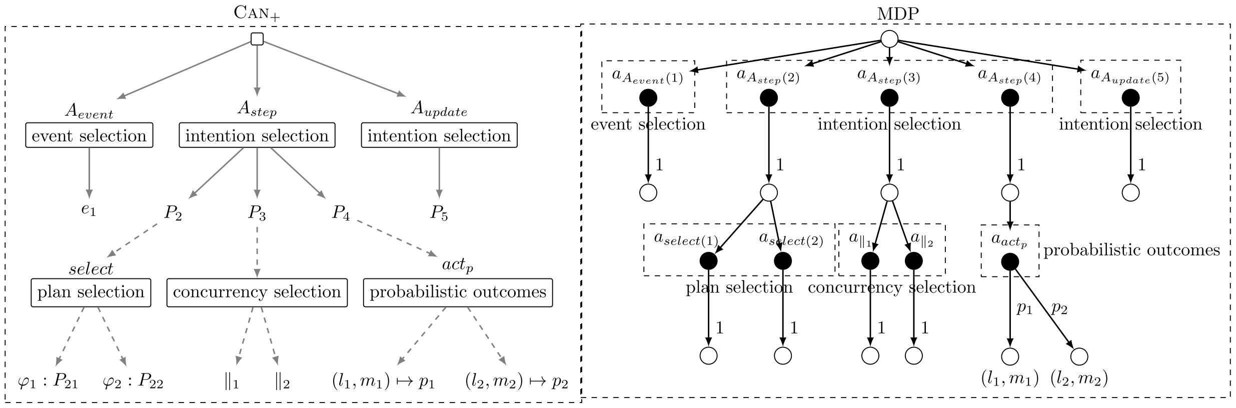


**Fig. 3.** Left: CAN$_+$ semantic rule possibilities highlighting event, intention, plan, and concurrency selection, probabilistic agent action outcomes, and uncertain beliefs. Solid lines are agent-level transitions and dashed lines are intention-level. Right: Corresponding MDP model of CAN$_+$ semantic rules with empty circles as states and solid circles as MDP actions.

of agents, we instead use probabilistic transitions $C \rightarrow_p C'$, i.e. this transition happens with probability $p$ [28]. For example, most (deterministic) CAN$_+$ semantics rules can, by default, be modified to be probabilistic rules with a probability 1. And the non-trivial use of probabilities primarily focuses on uncertain action outcomes of the agents (which will be introduced shortly in Section 4.1).

To translate a (probabilistic) semantic rule named *rule* (Eq. (1)) to an MDP action, we include an MDP action $a_{rule}$ in the set of all MDP action labels and define the transition function $\delta$ such that Eqs. (2) and (3) hold:

$$\frac{\lambda_1 \quad \lambda_2 \quad \cdots \quad \lambda_n}{C \rightarrow_p C'}\ rule \tag{1}$$

$$\delta(C, a_{rule}) \text{ is defined iff } \lambda_i \text{ holds in } C \text{ with } i \in \{1, 2, \cdots, n\} \tag{2}$$

$$\delta(C, a_{rule})(C') = p \tag{3}$$

Condition (2) says a transition of CAN$_+^m$ is only enabled if the transition would be enabled in CAN$_+$, i.e. the premises $\lambda_i$ of *rule* are all met. Condition (3) defines the probability of transitioning from $C$ to $C'$ in CAN$^m$ as the same as the probability of transitioning in CAN$_+^m$. The mapping of semantic rules to MDP actions is applied to both intention and agent-level rules from CAN.

The overview of our translation from CAN$_+$ to an MDP is depicted in Fig. 3. CAN$_+$ features non-deterministic transition, e.g. for plan selection and choices appear throughout both the agent and intention level transitions. Furthermore, agent actions have probabilistic outcomes sampled from a distribution and uncertain beliefs. The right-hand of Fig. 3 presents our MDP model of CAN$_+$ with translated MDP actions for each semantic rules. We detail this translation in the next sections.

## 4.1. Probabilistic action outcomes

Probabilistic transitions occur when we add support for probabilistic action outcomes for agents. In original CAN, the outcomes of an action are fixed outcomes when an agent action is executed. However, in practice agent actions often fail, e.g. there is a chance an agent tries to open a door but cannot. To capture these uncertain outcomes in agent actions in epistemic setting, we introduce a new *probabilistic* semantic rule $act_p$ where $\mu = [(l_1, m_1) \mapsto p_1, \ldots, (l_n, m_n) \mapsto p_n]$ is a user-specified outcome distribution where $(l_i, m_i)$ denotes an input of a compact epistemic state defined in Definition 1, $\mu((l_i, m_i)) = p_i$ and $\sum_{i=1}^{n} p_i = 1$.

$$\frac{act : \psi \leftarrow \mu \quad \mu((l_i, m_i)) = p_i \quad \mathcal{W} \models \psi}{\langle \mathcal{W}, act \rangle \rightarrow_{p_i} \langle \mathcal{W} \circ (l_i, m_i), nil \rangle}\ act_p$$

For mapping intention-level CAN$_+$ configurations to MDP states we use the fact that $\langle \mathcal{W}, P \rangle$ is a special case of $\langle E^e, \mathcal{W}, \Gamma \rangle$ where $\mathcal{W}$ is a compact epistemic state and $P \in \Gamma$ allowing us to translate the intention-level semantic rules to MDP actions according to the rule translation template in Eqs. (2) and (3). The probabilistic nature of $act_p$ is reflected in the MDP action $a_{act_p}$:

$$\delta(C, a_{act_p})(C') = p_i \text{ s.t. } C = \langle \mathcal{W}, act \rangle,\ act : \psi \leftarrow \mu, \mathcal{W} \models \psi,$$
$$\mu((l_i, m_i)) = p_i, \text{ and } C' = \langle \mathcal{W} \circ (l_i, m_i), nil \rangle$$

## 4.2. Intention-level semantics

The intention-level semantics (Fig. 1) specify how to evolve any single intention. Most rules have deterministic outcomes with the exception of some rules such as *select* (Fig. 1) which is non-deterministic, i.e. when we select a single applicable plan from the set of relevant plans. To use rules like this in CAN$_+^m$ we need to lift the non-determinism, hidden *within* the rules, to non-determinism *between* rules. We do this by introducing a new rule for each possible choice, e.g. a rule for each possible plan that can be selected. As notation, we describe this set of rules via parameterised rules, e.g. $select(n)$ as follows:

$$\frac{\langle n, \varphi : P\rangle \in \Delta \quad \mathcal{W} \models \varphi}{\langle \mathcal{W}, e : (\!|\, \Delta \,|\!)\rangle \rightarrow_1 \langle \mathcal{W}, P \rhd e : (\!|\, \Delta \setminus \{\langle n, \varphi : P\rangle\} \,|\!)\rangle}\ select(n)$$

where $n$ is an identifier for the plan and can be trivially assigned using positions in the plan library (i.e. $1 \leq n \leq |\Pi|$). Once we chose a plan rule, it is always successful ($p = 1$) and it can be similarly translated into an MDP action, denoted as $a_{select(n)}$, using the previous translation template.

### 4.3. *Agent-level semantics*

Agent-level CAN$_+$ rules (Fig. 2) determine how an agent responds to events and progresses/completes intentions. There are three rules and each has a non-deterministic outcome: $A_{event}$ that selects one event to handle from a set of pending events; $A_{step}$ that progresses one intention from a set of partially executed intentions; and $A_{update}$ that removes an unprogressable intention from a set of unprogressable intentions. As with $select$ in Section 4.2, to use these in the CAN$_+^m$ model, we need to move from non-deterministic rules to a set of deterministic rules parameterised by the outcome. The new rules are

$$\frac{\langle n, e\rangle \in E^e}{\langle E^e, \mathcal{W}, \Gamma\rangle \Rightarrow_1 \langle E^e \setminus \{\langle n, e\rangle\}, \mathcal{W}, \Gamma \cup \{\langle n, e\rangle\}\rangle}\ A_{event}(n)$$

$$\frac{\langle n, P\rangle \in \Gamma \quad \langle \mathcal{W}, \langle n, P\rangle\rangle \rightarrow_p \langle \mathcal{W}', \langle n, P'\rangle\rangle}{\langle E^e, \mathcal{W}, \Gamma\rangle \Rightarrow_p \langle E^e, \mathcal{W}', (\Gamma \setminus \{\langle n, P\rangle\}) \cup \{\langle n, P'\rangle\}\rangle}\ A_{step}(n)$$

$$\frac{\langle n, P\rangle \in \Gamma \quad \langle \mathcal{W}, \langle n, P\rangle\rangle \nrightarrow_1}{\langle E^e, \mathcal{W}, \Gamma\rangle \Rightarrow_1 \langle E^e, \mathcal{W}, \Gamma \setminus \{\langle n, P\rangle\}\rangle}\ A_{update}(n)$$

Event parameters are specified by numbering them based on an ordering on the full set of events, e.g. $\langle n, e\rangle$ with $n \in \mathbb{N}^+$ as an identifier. We identify (partially executed) intentions based on the identifier of the top level plan that led to this intention, e.g. for $P \in \Gamma$ we assign a label $n \in \mathbb{N}^+$ that is passed alongside the intention. This style of labelling assumes only one instance of an event can be handled at once (this is enough to imply the top level plans are also unique). As with $select$ the transition probability is 1 in the cases of $A_{event}(n)$ and $A_{update}(n)$ as the rule, if selected, always succeeds. The (omitted) MDP actions for rules $A_{event}(n)$ and $A_{update}(n)$ can be similarly given as $a_{A_{event}(n)}$ and $a_{A_{update}(n)}$, respectively. The rule $A_{step}(n)$ says that agent-level transitions depend on the intention-level transitions and we need to account for this in the transition probabilities.

## 5. Bigraph encoding of CAN$_+^m$ model

We have shown in previous work [26] that we can describe the original semantics of CAN (without the support of uncertain beliefs) as an MDP. In this section, we show how we represent uncertain beliefs as epistemic state encoded in bigraphs and how it connects with the existing bigraph encoding introduced in [26]. We begin with a brief introduction to bigraphs.

### 5.1. *Bigraphs*

Bigraphs are a universal graph-based modelling formalism introduced by Milner [19], with conditional, priority, parameterised, and probabilistic extensions [29,20]. Bigraphs have an equivalent algebraic and diagrammatic form, and we use the diagrammatic form here.

An example bigraph is in Fig. 4a. It consists of a set of entities, e.g. A, B, drawn as (coloured) shapes.[4] Entities can be related through *nesting* (to arbitrary depth), e.g. the B entities inside A. Entities can also be related through hyperlinks (permitting any-to-any links rather than just one-to-one as is usual), such as the green link between the B and C entities. Entities have a *fixed* number of links, called the arity, although a link can be disconnected as shown by the C entity in Fig. 4c. The filled grey rectangles denote that other (unspecified) entities can exist here. Dashed unfilled rectangles are *regions* that represent parallel parts of the system: that is, these two regions can, but do not have to, share a single parent in some larger system model.

A bigraph represents a system at a single point in time. To allow models to evolve over time we can specify *reaction rules* of the form $L \longrightarrow R$, where $L$ and $R$ are bigraphs. An example reaction rule is in Fig. 4b, which models the disconnection of B and C and also removes the nesting of B in A. The filled grey rectangles are called *sites* and represent parts of the model, below some entity, that have been abstracted away. That is, it allows matching on an A with *multiple* children. Without the site, the rule would only match when A had a single B child. Similarly, the use of the name $x$ means that it is open which allows B to be connected elsewhere during the match (in this case the other B). If there was no name (closed), it would only match exactly one B connected to one C. Reaction rules can affect both linking and placement, as shown here with the B entity also moving next to C. Unlike some process calculi, Bigraphs do not have a fixed number of entities and it is possible to add and remove entities[5] as the model evolves, e.g. you could define a rule that adds a new B to an A node.

[4] We often use the shape to denote the entity type to reduce the need for excessive labelling.

[5] But not new entity types.

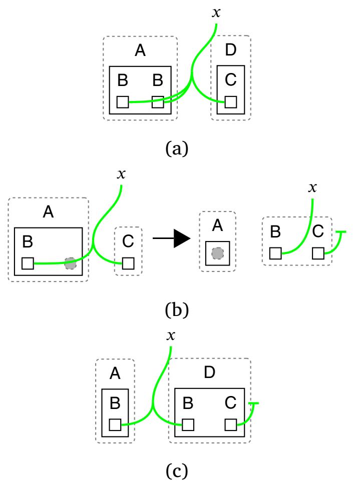
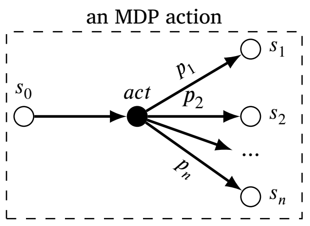


**Fig. 4.** (a) Example bigraph, (b) reaction rule, and (c) result after applying (b) to (a). (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)



a (non-empty) set of reaction rules in bigraphs

$$act = \{\mathtt{r}_1 = L_0 \xrightarrow{w_1} R_1, \mathtt{r}_2 = L_0 \xrightarrow{w_2} R_2, \cdots, \mathtt{r}_n = L_0 \xrightarrow{w_n} R_n\}$$

where $L_0$ encodes state $s_0$, $R_i$ encodes $s_i$ $(i = 1, \cdots, n)$, and $\dfrac{w_i}{\sum_{j=1}^{n} w_j} = p_i$

**Fig. 5.** Left: MDP action $act$ applying to state $s_0$ with a probability $p_i$ reaching to the state $s_i$ $((i \in \{1, \cdots, n\})$. Right: corresponding bigraph reaction rules to encode $act$.

Conditional bigraphs [29] (which we use here) allow extra constraints on rule application by enforcing that a specific bigraph occurs (or does not occur) within the sites[6] of a match. The syntax of a condition is: if $\langle -, \boxed{\overset{\mathsf{B}}{\square}}, \downarrow \rangle$, where the minus sign denotes enforces absence a bigraph B, and the down arrow indicates we are checking the sites. An example conditional rule is in where the condition enforces that T cannot appear in the site.

Bigraphs allow reaction rules to be weighted, e.g. $\mathtt{r} = L \xrightarrow{3} R$ and $\mathtt{r}' = L \xrightarrow{1} R'$, such that if both (and only) $\mathtt{r}$ and $\mathtt{r}'$ are applicable then $\mathtt{r}$ is three times as likely to apply as $\mathtt{r}'$. The normalisation will be applied automatically by the BigraphER tool [24] (an open-source language and toolkit for bigraphs) to obtain the specific probability. Non-deterministic choices (e.g. an MDP action) can be modelled as a non-empty set of reaction rules. For example, we can have an MDP action $\mathtt{a} = \{\mathtt{r}, \mathtt{r}'\}$ and once it is executed, it has a distribution of 75% transition from $L$ to $R$ and 25% from $L$ to $R'$. Fig. 5 depicts how to encode any MDP action in bigraphs. To execute and, importantly, analyse our bigraph model, we employ BigraphER to exhaustively explore all possible behaviours of the model to capture the transition systems of an MDP, and states may be labelled using bigraph patterns that assign a state *predicate* label if it contains (a match of) a given bigraph. The labelled MDP transition systems can be then exported for quantitative analysis and strategy synthesis in PRISM and Storm by importing the underlying MDPs produced by BigraphER. We reason about the minimum or maximum values of properties such as $\mathcal{P}_{max=?}\mathbf{F}[\phi]$ in Probabilistic Computation Tree Logic (PCTL) [30]. $\mathcal{P}_{max=?}\mathbf{F}[\phi]$ expresses the maximum probability of $\phi$ holding *eventually* in all possible resolutions of non-determinism.

## *5.2. Bigraph encoding of epistemic states*

Recall that a compact epistemic state $\mathcal{W}$ is a mapping $\mathcal{W} : \mathcal{A}t \to (\mathbb{Z}, \mathbb{Z})$ such that $\mathcal{W}(a) = (\mu_+, \mu_-)$ where $\mathcal{A}t$ is a set of all possible belief atoms. To model this in bigraphs, we use nesting to associate each atom with a positive and negative value, as shown graphically in Fig. 6. We represent $\mu_+$ and $\mu_-$ as *parameterised* entities, PositiveValue(m) and NegativeValue(n), respectively, and then nest them within the entity that represents the belief atom (highlighted in red). Finally, the entire bigraph is then further nested in the belief base to show it is a member of a belief base. The site represents the application of this compact epistemic state to other atoms, i.e. $\mathcal{W}(b)$ where $b \in \mathcal{A}t \setminus \{a\}$.

[6] More generally we allow checking both sites, and wider context, e.g. the parents of a bigraph match.

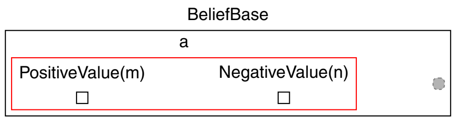


**Fig. 6.** Bigraph representation of a compact epistemic state for a belief atom $\mathcal{W}(a) = (\mu_+, \mu_-)$ where we nest the parameterised bigraph entities PositiveValue(m) and NegativeValue(n), representing $\mu_+$ and $\mu_-$ respectively, under the entity a for the belief atom $a$.

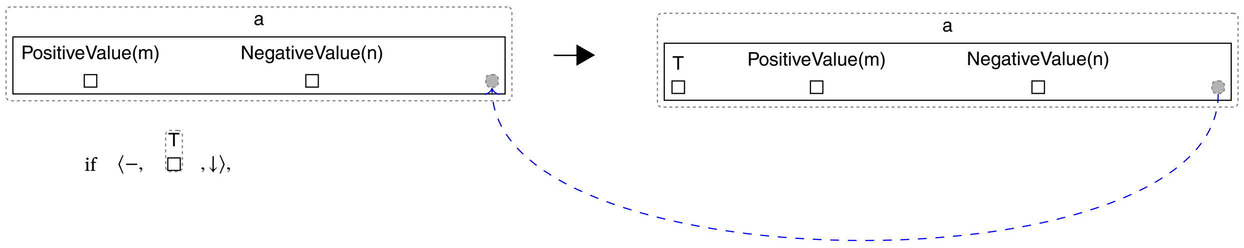


**Fig. 7.** Conditional reaction rule for the entailment for a compact epistemic state for a belief atom with PositiveValue(m) and NegativeValue(n) if no truth entailment (though symbol − indicating that the bigraph of T, i.e. the truth entailment, should *not* appear/be matched) and $m > n$. This condition ensures we do not get duplicate T entities. The dashed arrow (called the instantiation map) forces the site on the right to be the copy of the site on the left.

Now, we introduce the parameterised reaction rule encoding the comparison between an atom's positive and negative values. Recall that we say an atom is believed to be true by $\mathcal{W}$, denoted as $\mathcal{W} \vDash a$ if $\mathcal{W}(a) = (\mu_+, \mu_-)$ and $\mu_+ > \mu_-$. Fig. 7 says that if the value of m in PositiveValue(m) is greater than the value of n in NegativeValue(n), the atom $a$ will be nested a token T in bigraph to represent its truth entailment.

## 5.3. *Quantitative analysis and strategy synthesis*

We revisit the robotic submarine example from the introduction to show how our approach provides uncertainty modelling, quantitative verification, and strategy synthesis. This example is inspired by the SUAVE exemplar [31,32].

### 5.3.1. *Example*

A robotic submarine is tasked with finding and inspecting a pipeline located on a seabed, and then sending the inspection results to human operators. There are many uncertainty factors: changes in water visibility, potential thruster failure, and unreliability of information transmission in seawater. While we consider a simplified implementation it is sufficient to demonstrate the core features of our approach, e.g. uncertainty modelling, quantitative verification, and strategy synthesis. Though we only give details of a single case study, users of the executable semantics can employ BigraphER to "run" models with different settings, e.g. external events, plan libraries, customised uncertain beliefs, and probabilistic actions.

For each part of the mission (scanning, surveying, sending), the submarine can choose to operate at three different sea depths: *low*, *medium*, and *high*. The depths have different effects depending on the part of the mission. For example, high depths allow the robotic submarine to have better visibility (i.e. the distance in meters within which the submarine can perceive objects) and this increases the chance of finding the pipeline during search. The probability of a thruster failure increases at high depths because, e.g. seaweed might wrap around the thrusters. Once the inspection is complete, the submarine transmits the collected results to the human operators. In this example, we made certain modelling assumptions: first, we assume that the thruster can only fail during the survey phase, as this requires active movement, unlike the scanning and reporting phases. Additionally, we assume that when the submarine attempts to maintain a specific depth, it does so successfully, meaning that depth is treated as a choice rather than an explicit belief.

The agent program for this scenario is in Fig. 8. A short commentary is as follows. Fig. 8a gives the initial uncertain beliefs on the conditions in which the robotic submarine is situated. For example, the (compact epistemic) state $\mathcal{W}(thruster\_functional) = (3, 1)$ shows there is more confidence the thruster is functional than not, and $\mathcal{W}(pipe\_found) = (0, 3)$ means it is likely the pipe is not yet found. Trialling different weights (which could, for example, be extracted from historical data) is possible by simply changing them in the compact epistemic state. In the next section, we will show how to support the automation of this entire process, from modelling to analysis, through CAN-VERIFY.

Plans for the submarine are given in Fig. 8b. For example, the first plan handles the overall mission $inspect\_pipe$ which is applicable when the agent believes the thruster is functional. To achieve the mission $inspect\_pipe$, the robotic submarine must achieve three internal events (i.e. sub-goals) in sequence (i.e. $find\_pipe$; $survey\_pipe$; $report\_back$). For these events, there are three plans representing the different depths that can be used for each sub-mission. For example, plans 2–4 allow the submarine to scan for the pipe at either low, medium, and high depths. Finally, Fig. 8c gives the action descriptions. For example, action

$$\mathcal{W}(thruster\_functional) = (3, 1)$$
$$\mathcal{W}(pipe\_found) = (0, 3)$$
$$\mathcal{W}(report\_sent) = (0, 2)$$

(a) Uncertain beliefs for submarine in a BDI agent.

1. $inspect\_pipe : thruster\_functional \leftarrow find\_pipe; survey\_pipe; report\_back.$
2. $find\_pipe : thruster\_functional \leftarrow scan\_low.$
3. $find\_pipe : thruster\_functional \leftarrow scan\_med.$
4. $find\_pipe : thruster\_functional \leftarrow scan\_high.$
5. $survey\_pipe : pipe\_found \leftarrow survey\_low.$
6. $survey\_pipe : pipe\_found \leftarrow survey\_med.$
7. $survey\_pipe : pipe\_found \leftarrow survey\_high.$
8. $report\_back : thruster\_functional \ \& \ pipe\_found \leftarrow report\_low.$
9. $report\_back : thruster\_functional \ \& \ pipe\_found \leftarrow report\_med.$
10. $report\_back : thruster\_functional \ \& \ pipe\_found \leftarrow report\_high.$

(b) Plans for submarine in a BDI agent.

$$scan\_low : true \leftarrow [(pipe\_found, 3) \mapsto 0.4, (\neg pipe\_found, 1) \mapsto 0.6]$$
$$scan\_med : true \leftarrow [(pipe\_found, 4) \mapsto 0.6, (\neg pipe\_found, 0) \mapsto 0.4]$$
$$scan\_high : true \leftarrow [(pipe\_found, 5) \mapsto 0.9, (\neg pipe\_found, 1) \mapsto 0.1]$$
$$survey\_low : true \leftarrow [(thruster\_functional, 3) \mapsto 0.9, (\neg thruster\_functional, 7) \mapsto 0.1]$$
$$survey\_med : true \leftarrow [(thruster\_functional, 1) \mapsto 0.8, (\neg thruster\_functional, 5) \mapsto 0.2]$$
$$survey\_high : true \leftarrow [(thruster\_functional, 2) \mapsto 0.6, (\neg thruster\_functional, 6) \mapsto 0.4]$$
$$report\_low : true \leftarrow [(report\_sent, 3) \mapsto 0.9, (\neg report\_sent, 2) \mapsto 0.1]$$
$$report\_med : true \leftarrow [(report\_sent, 3) \mapsto 0.8, (\neg report\_sent, 2) \mapsto 0.2]$$
$$report\_high : true \leftarrow [(report\_sent, 3) \mapsto 0.7, (\neg report\_sent, 2) \mapsto 0.3]$$

(c) Probabilistic actions in plans in Fig. 8b.

**Fig. 8.** Agent programs for a robotic submarine.

Listing 1: A list of properties with its associated value for the robotic submarine where mission_success denotes the event *inspect_pipe* successfully being processed, and thruster_functional, pipe_found, and report_sent denote the corresponding belief holding true.

1 $\mathcal{P}_{min=?}\mathbf{F}[\,\text{mission\_success}\,]$ (value 0)
2 $\mathcal{P}_{max=?}\mathbf{F}[\,\text{mission\_success}\,]$ (value 0.81)
3 $\mathcal{P}_{max=?}\mathbf{F}[\,\text{pipe\_found} \wedge \text{thruster\_functional} \wedge \text{report\_sent}\,]$ (value 0.73)

$scan\_high : true \leftarrow [(pipe\_found, 5) \mapsto 0.9, (\neg pipe\_found, 1) \mapsto 0.1]$ says that the action $scan\_high$ has a 90% chance of strengthening the belief of $pipe\_found$ by the value of 5 or a 10% chance of $\neg pipe\_found$ by the value of 1.

### 5.3.2. Quantitative analysis

For analysis, we label states where properties of interest hold. We use mission_success to denote the event *inspect_pipe* is successfully addressed by the robotic submarine. The labels of thruster_functional, pipe_found, and report_sent denote the states where the corresponding belief are entailed to be true. A full list of properties checked for this example is in Listing 1.

Property $\mathcal{P}_{min=?}\mathbf{F}[\,\text{mission\_success}\,]$ checks the minimum probability of the event *inspect_pipe* being processed successfully over all possible adversaries. This property returns a value of 0, meaning there is a possible situation where the robot fails to complete the mission. In this case, it could be that the robotic submarine scanned the seabed at too low depths and missed the pipe. Property $\mathcal{P}_{max=?}\mathbf{F}[\,\text{mission\_success}\,]$ determines the best possible outcome to address the event *inspect_pipe* and returns a value of 0.81.[7] Besides looking at the overall mission success of the agent program, we can also investigate the reachability of beliefs of interest. For example, in this case, we may want to know the maximum probability of the robotic submarine eventually believing it has found the pipe, successfully sent the results, and still has a functioning thruster, i.e. $\mathcal{P}_{max=?}\mathbf{F}[\,\text{pipe\_found} \wedge \text{thruster\_functional} \wedge \text{report\_sent}\,]$. Interestingly, we notice that the value of property 3 is smaller than the value of property 2. The reason is that an agent program is regarded as completed if it can successfully be executed but does not necessarily care about the certain beliefs that may occur due to it, e.g. uncertain effects of actions. For example, as long as the robotic submarine can execute action *report_low*, it regards the internal event *report_back* as achieved, hence the event *inspect_pipe* is also achieved. But action *report_low* still has some probability, resulting in the report not being successfully transmitted to the human operators. This example highlights the importance of being able to analyse a richer sets of reachability properties.

[7] This probability is never 1 as there is always a chance of failing regardless of the depth the submarine chooses.

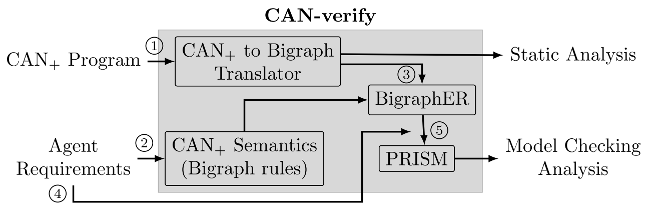


**Fig. 9.** Toolchain overview: ① agent program compilation to bigraphs, ② predicate labelling in bigraph model, ③ (exhaustive) execution of programs, ④ built-in and user-defined belief-based specification formalisation in PCTL, ⑤ formal verification.

### 5.3.3. *Strategy synthesis*

Properties to be model checked on MDPs usually quantify over strategies (or policies) of the model, i.e. over the different possible ways that nondeterminism can be resolved in the model. For example, property $\mathcal{P}_{max=?}\mathbf{F}[\,\mathsf{pipe\_found} \wedge \mathsf{thruster\_functional} \wedge \mathsf{report\_sent}\,]$ determines the maximum probability, over all strategies, of reaching a state in which these beliefs are believed to be true. When checking such properties, you can also ask PRISM to generate a corresponding (optimal) strategy, which yields this maximum probability when followed. PRISM generates several types of strategies. The simplest are memoryless deterministic strategies, which pick a single action in each state. Here, the optimal adversary instructs the robotic submarine to scan at high depth to get a better vision capability to find the pipe, survey at low depth to avoid getting trapped by the seaweeds and causing thruster failure, and send the results at low depth to ensure a higher chance of information transmission.

## 6. CAN-VERIFY

To widen access to formal methods without requiring users to be familiar with complex encoding and verification techniques, we extended our automated tool, CAN-VERIFY [27], for BDI programmers to verify the MDP models of epistemic-enabled CAN agents i.e. CAN$_+$. CAN-VERIFY takes as input CAN$_+$ programs which supports epistemic beliefs, and supports the following features:

- syntax checking for CAN$_+$ programs;
- a symbolic interpreter for CAN$_+$ programs;
- exhaustive exploration, i.e. symbolically taking all possible computation paths, of CAN$_+$ programs;
- 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 structured natural language;
- verification of agents parameterised by their initial belief base to support analysis of agent behaviours under different initial environments.

Our tool utilises the bigraph encoding of the MDP model of a CAN agent with epistemic beliefs (i.e. CAN$_+$), and runs model checking tools as required. The dataflow of the toolchain is in Fig. 9. Step ① translates input CAN$_+$ into bigraphs expressed in the BigraphER [33] language according to Section 5. During the translation, static checks are performed and errors/warnings reported to users. 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 in Section 4, and asks BigraphER to explore all possible executions. The output of BigraphER (an explicit MDP transition system with state predicate labels), and built-in and user-defined temporal logic formulae (complied from ④) are then verified by PRISM[8] (⑤). Next, we will go through the features of our tool in detail one by one. The source code and the running example of robotic submarine shown in this work, are openly available.[9]

### 6.1. *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 minimum/maximum number of plans for an event.

### 6.2. *CAN$_+$ interpreter*

As the bigraph model includes the semantics for the CAN$_+$, 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.

[8] Any model checker supporting explicit model import would work. PRISM is chosen as BigraphER natively supports PRISM format.
[9] https://github.com/Mengwei-Xu/Can-verify.

Listing 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 Pmax=? [ F ("no_failure" & X ("empty_intention")) ];
2 Pmin=? [ F ("no_failure" & X ("empty_intention")) ];
3 Pmax=? [ F ("failure" & X ("empty_intention")) ];
4 Pmin=? [ F ("failure" & X ("empty_intention")) ];
5 Pmax=? [ F (<belief-list>) ];
6 Pmin=? [ F (<belief-list>) ];
```

Listing 3: CAN agent for robotic submarine.

```
 1 // Initial belief bases
 2 1. thruster_functional : <3, 1>, pipe_found : <0, 3>, report_sent: <0, 2>
 3 2. thruster_functional : <3, 4>, pipe_found : <0, 3>, report_sent: <0, 2>
 4 // External events
 5 inspect_pipe : 1
 6 // Plan library
 7 1 : inspect_pipe : thruster_functional <- find_pipe; survey_pipe; report_back.
 8 2 : find_pipe : thruster_functional <- scan_low.
 9 3 : find_pipe : thruster_functional <- scan_med.
10 4 : find_pipe : thruster_functional <- scan_high.
11 5 : survey_pipe : pipe_found <- survey_low.
12 6 : survey_pipe : pipe_found <- survey_med.
13 7 : survey_pipe : pipe_found <- survey_high.
14 8 : report_back : thruster_functional & pipe_found <- report_low.
15 9 : report_back : thruster_functional & pipe_found <- report_med.
16 10 : report_back : thruster_functional & pipe_found <- report_high.
17 // Actions description
18 scan_low : true<- <pipe_found: {3,4}>,<pipe_found: {1,6}>
19 scan_med : true<- <pipe_found: {4,6}>,<pipe_found: {0,4}>
20 scan_high : true<- <pipe_found: {5,9}>,<pipe_found: {1,1}>
21 survey_low : true<- <thruster_functional: {3,9}>,<thruster_functional: {7,1}>
22 survey_med : true<- <thruster_functional: {1,8}>,<thruster_functional: {5,2}>
23 survey_high : true<- <thruster_functional: {2,6}>,<thruster_functional: {6,4}>
24 report_low : true<- <report_sent: {3,9}>,<report_sent: {2,1}>
25 report_med : true<- <report_sent: {3,8}>,<report_sent: {2,2}>
26 report_high : true<- <report_sent: {3,7}>,<report_sent: {2,3}>
```

## 6.3. Model checking of BDI programs

Model checking is enabled by taking the executable model of the agent program and constructing a labelled transition system in MDP format of the program's possible executions that allow checking agent requirements against this model. Built-in agent requirements for generic properties include determining the maximum/minimum probability that an event finishes with failure or success (these failure or success states are labelled using bigraph predicates [34] automatically). These requirements are translated into probabilistic branching time temporal logic formulae e.g. Probabilistic Computation Tree Logic (PCTL) [35,30] for the PRISM model checker. Example requirements are in Listing 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. the maximum/minimum probability of some desired/avoided beliefs would hold true in all possible agent behaviours. To avoid requiring users to formalise these properties in PCTL syntax, properties are instead expressed in natural language. For example, the input of "What is the maximum probability that eventually the belief `actual_belief` holds?"[10] corresponds to the PCTL formula `Pmax=?[F("actual_belief")]`. 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 [36]. 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 $\text{CAN}_+$ input. This is useful as users do not have to run the agent with each initial belief manually and 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.

## 6.4. Example

We now re-visit the robotic submarine again given in Fig. 8 and show how to model check it in CAN-VERIFY without the prerequisite of formal modelling and logic. Recall CAN-VERIFY takes an input of a BDI program written in the language of $\text{CAN}_+$. An agent design for it is in Listing 3. The external event `inspect_pipe` (line 4) initiates the inspection mission, whereas line 2 gives two possible

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

**Table 1**
States, transitions, construction time (s), and verification time (s) for property $\mathcal{P}_{max=?}\mathbf{F}[\,\mathsf{pipe\_found} \wedge \mathsf{thruster\_functional} \wedge \mathsf{report\_sent}\,]$.

| | |
|---|---|
| States | 211 |
| Transitions | 35 |
| Transition system construction time (s) | 170.10 |
| Property verification time (s) | 0.73 |

Listing 4: CAN agent for drone wetland surveillance.

```
 1 // Initial belief bases
 2 1. patch1_flood : <0, 0>, status_patch1 : <0, 0>
 3 // External events
 4 survey_patches : 1
 5 // Plan library
 6 1 : survey_patches : true <- survey_patch1; survey_patch2.
 7 2 : survey_patch1 : true <- photo_patch1; report_patch1.
 8 3 : survey_patch2 : true <- photo_patch2; report_patch2.
 9 // Actions description
10 photo_patch1 : true<- <patch1_flood: {2,8}>,<patch1_flood: {2,2}>
11 report_patch1 : true<- <status_patch1: {5,7}>,<status_patch1: {5,3}>
12 photo_patch2 : true<- <patch2_flood: {2,8}>,<patch2_flood: {2,2}>
13 report_patch2 : true<- <status_patch2: {5,6}>,<status_patch2: {5,4}>
```

initial uncertain beliefs to allow parameterised model checking against different initial conditions. Lines 7-16 are the exactly replicates of the plans given in Fig. 8. Finally, lines 18-26 are the description of actions, and we have adapted the syntax to allow easy parsing. In particular, we notice that instead of giving exactly the final probability of the action effects, we rely on the automatic normalisation provided in CAN-VERIFY. For example, we have

```
<pipe_found: {5,9}>,<pipe_found: {1,1}>
```

is equivalent to

$$[(pipe\_found, 5) \mapsto 0.9, (\neg pipe\_found, 1) \mapsto 0.1]$$

By default, CAN-VERIFY will analyse properties such as what is the minimum/maximum probability of an event being eventually addressed successfully. To model check belief-based properties, for example, $\mathcal{P}_{max=?}\mathbf{F}[\,\mathsf{pipe\_found} \wedge \mathsf{thruster\_functional} \wedge \mathsf{report\_sent}\,]$, we can use the following structured natural language stored in a `.txt` file:

```
What is the maximum probability that eventually the beliefs (pipe_found, thruster_functional,
report_sent) hold
```

CAN-VERIFY will output the same results as shown in Listing 1.

Performance measurements gathered when checking this property are in Table 1 and a dedicated scalability experiment is in the next section. Most time is spent in the construction of transition system (as expected) and computing the probability of a PCTL property is significantly faster.

### 6.5. *Scalability*

We provide empirical insights on the approach, focusing on the scalability of model construction in the underlying bigraph formalism. The results are obtained on a laptop with a 16-core Intel Core i7-11800H at 2.30 GHz (hyperthreaded), 16 GB memory, and running 64-bit Ubuntu Linux 24.04 LTS. Instructions on how to reproduce this experiment can be found in the model repository.[11] For analysis we use the following scenario based on wetland surveillance. An autonomous drone is tasked to check, after a storm, whether each patch of land is flooded or not, and subsequently report the land status to human operators at the base. The drone is equipped with high-resolution cameras and Machine Learning (ML) algorithms to analyse the captured photos. The employment of ML inevitably introduces uncertainty to the analysis of the photos. For example, the drone may mistake a mountain pond as a flood. As the drone flies further from the base, the remote condition of the wetland may impact the radio transmission, i.e. the messages sent from the drone may fail to reach back to the base. We capture these simplified behaviours of drone wetland surveillance to survey two patches of land one by one in Listing 4. Line 2 in Listing 4 gives the initial belief base of the drone. The drone is set to address the external event $\mathsf{survey\_patches}$ (line 4). In this case, the drone needs to survey two patches one by one, i.e. $\mathsf{survey\_patch1}; \mathsf{survey\_patch2}$.

[11] https://github.com/Mengwei-Xu/Can-verify/tree/main/Scalability.

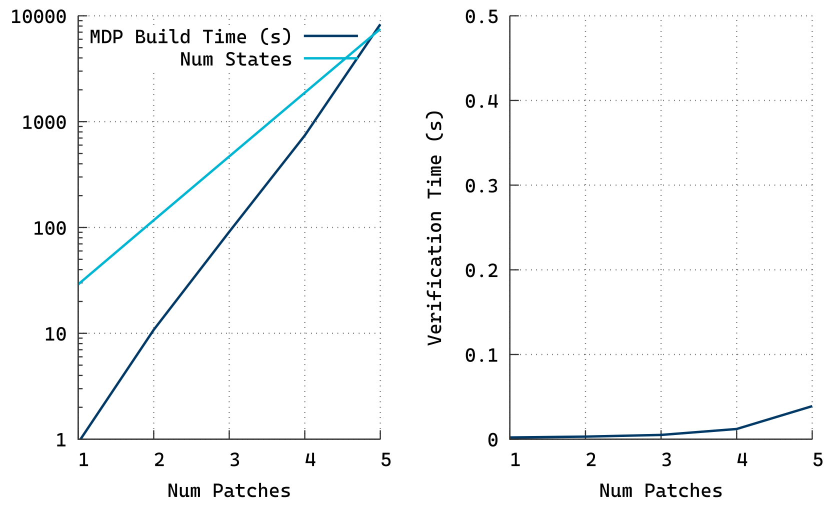


**Fig. 10.** Time to build the MDP, and states, increase exponentially (note log scale). Verification time almost constant and negligible.

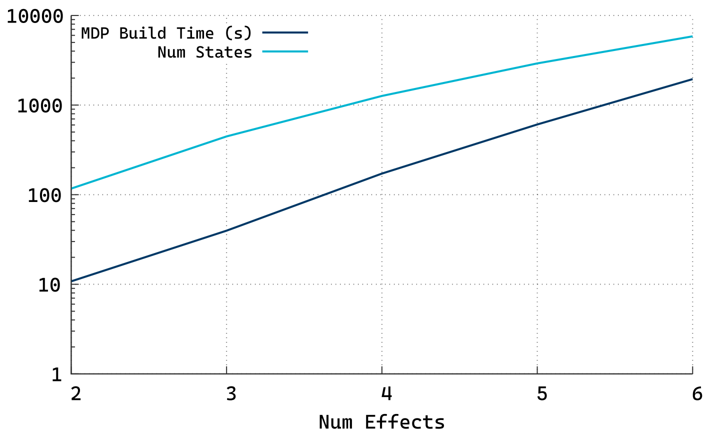


**Fig. 11.** Transition system construction time and states increase exponentially with number of effects.

Lines 2-3 instruct the drone to photograph the land patch and report the results computed by the ML subsystems. The design of such a drone from any number of patch surveillance can be similarly given. To evaluate scalability, we increase the number of patches to be surveyed.

Fig. 10 gives the time to build the transition system (MDP) and the number of states as the number of patches increases. As expected, because we explore all possible paths, there is an exponential increase in time from a couple of seconds to a couple of hours. The intersection of states and build time suggests the matches needed for bigraph rewriting are getting more complex, likely due to introducing symmetries in the states structure. New solving technologies could improve on this in future and without requiring changes from the agent programmer. The same pattern is not observed in the verification time, that remains negligible (under 0.01 s) even as the states increase. Finally, to further evaluate the proposed approach against the computational effects of a number of outcomes per action, we extend the design of a drone for two patches with an increasing number of outcomes for all actions, e.g. with different strengths to increase the positive value of a belief in Listing 4. Fig. 11 shows a similar (expected) exponential increase in build time and number of states. This time the states increase much slower, and the matches themselves do not appear to be getting significantly more complex (as the lines are growing at similar rates).

## 7. Reflection

In this section, we reflect on the insights gained by encoding the epistemic states in bigraph and constructing an epistemic-enabled extension of CAN language, including the process of building bigraph models. We detail our first-hand experience of the value and limits of the bigraph approach applied to agent languages and their policies, which is not included in our previous work e.g. [26].

By building on an existing encoding of CAN in bigraphs [26], much of the epistemic extension required limited effort. In fact, most bigraph encoding extensions are primarily about how to encode the epistemic states separately and then connect them with the rest of the encoding in [26]. This high modularity benefits the nature of bigraph which uses parallel regions to separate models into different, but interacting, perspectives. For example, the epistemic states mainly affect the belief side of encoding. As such, we can separate this belief encoding without worrying about other aspects of the agent. Once it is completed, we can then move on to the interaction part of encoding between beliefs and all necessary parts of the rest (which is mainly on entailment).

Besides the parallel nature of the bigraph modelling, we also want to highlight that the graphical nature of bigraphs as seen e.g. in Figs. 6 and 7 provided a highly visual modelling and encoding experience and enabled us to locate the bugs with ease during the model development. The current encoding for the entailment of the epistemic states suffers from code redundancy as bigraphs do not natively support the comparison of two values as the condition of the reaction rule. Our current approach is to fix one parameter (either $\mu_+$) and parameterise the other. That said, with the use of the CAN-VERIFY tool, the users will not have the need to worry about any of these technical issues.

For verification, unlike other approaches in BDI languages, we are the first ones to be able to provide the strategy synthesis. As such, our approach can give agent designers an indication of the type of strategies that may be needed for a given application. However, the current CAN-VERIFY is unable to provide the automation for this yet. This is largely due to the complex task of presenting the users with a friendly strategy (mapping states to actions) without showing them the necessary underlying bigraph-related representation.

We also note that our approach is able to provide reward-based verification (as we have already shown in our previous [26] with Storm for reward-based synthesis). But we decided not to show in this paper. One of the reasons is that it is actually very difficult to reason about the accumulated rewards for reaching some target set of states if these states cannot be eventually reached with probability 1. It is due to a choice that both PRISM and Storm made when designing the reward property specification. They assume that if there is a non-zero probability of not reaching the target state (i.e. the probability of reaching it is less than 1), it is reasonable to say the path continues indefinitely without reaching the target state (i.e. the overall expected reward for being infinite). This severely limits the usage of probabilistic actions, which are crucial in real scenarios. A potential solution is to use the state reward (a certain amount of rewards is assigned if a certain state holds). Then the reward information can be specified as a temporal formula in property specifications. (e.g. what is the maximum probability of reaching this state which gives some certain reward). Unfortunately, this makes modelling and reasoning more cumbersome, and future work is required to investigate this.

Finally, we point out that our approach of extending the state of MDPs with epistemic states is a deliberate choice away from Partially observable Markov decision processes (POMDPs) which have been widely used for decision-making and verification under uncertainty (detailed related work will be given in the next section). The reason includes two considerations. The first and fundamental one is that we believe that the uncertainty of the beliefs is not necessarily to be probabilistic with exact probability (e.g. varying levels of confidence) or is difficult to quantify probabilistically, such as ambiguity. The second consideration is the computational complexity associated with POMDPs, which often makes them impractical for large-scale problems. By utilising epistemic states, we aim to provide a more scalable and conceptually intuitive framework for handling uncertainty in decision-making processes. Meanwhile, by having epistemic states in MDPs, we not only manage to obtain more modelling expressiveness from epistemic logic, but still being able to use PCTL for analysis which is generally more efficient compared to epistemic logic [37].

## 8. Related work

Optimal decision-making under uncertainty is a core problem in Artificial Intelligence (AI). A prime example is planning [13,14]: studying how to find good or optimal strategies to maximise rewards or the probability of reaching a goal and MDPs are also used as a fundamental mathematical models for planning. Formal verification coincides with planning when formulas in temporal logic express reachability goals (i.e. a set of final desired states) and verification methods are used to extract a particular evolution of the system that makes temporal formulas true. That is, verification focuses on checking if (reachability) properties hold for a system and obtaining strategies is a side effect. Our aim is not to compete with AI planning, but to use planning-like benefits in our verification framework for BDI agents. A prominent sub-field for finding good strategies is through reinforcement learning (RL) [38]. RL automatically trains agents to take actions to maximise a reward in an uncertain environment. Here, a concise specification of an MDP (capturing both the agent and the environment) is executed in an initially random manner and over time RL improves the reward of every state-action pair executed to yield good strategies. There has been promising work unifying planning, learning and verification [39].

Partially observable Markov decision processes (POMDPs) provide a general framework for decision-making [40] and verification [41] under probabilistic beliefs. Given the computational complexity of POMDPs, the typical solution to it is through Monte-Carlo sampling e.g. in [42] for planning. Meanwhile, the work [43] proposed a grid-based abstraction of the uncountable belief space induced by partial observability for verification.

The BDI community is interested in the non-determinism e.g. in the selection of event, plan and intention selection. And often this is usually done through modifying or replacing the original BDI reasoning entirely with other decision-making techniques. Although most BDI agent languages specify selection choices (e.g. plan selection) made by the agent in non-deterministic fashion, it is typical in practice to constrain the non-determinism through manual ordering—either statically [5] or at run-time [44]—to enforce simple deterministic behaviours. Since manual ordering can be restrictive, some selection strategies are given by using advanced planning algorithms [45,46]. For example, in [47] agent programs are compiled to TÆMS framework to represent the coordination relations e.g. “enables” and “hinders” between tasks and employ the Design-To-Criteria scheduler for intention selection. Other works show many of the intention progress issues can be modelled as AI planning problems and resolved through suitable planners [48]. It is not a new idea to integrate advanced decision-making techniques into BDI. There is a large body of work [45] to employ planning to synthesise new plans to achieve an event when no pre-defined plan worked or exists. For example, work [49] shows how the integration of planning and BDI can be done at the semantic level. Our approach not only ensures the safety of agent behaviours through formal verification, but also the quality of agent decision-making through optimal adversary generation.

An increasingly popular topic is intention progression [50], e.g. the contest [51], that helps the agent to make better decisions on event/plan/intention selection. Uncertain beliefs are also an important part of the research community. For example, the work [23] provides the capability to BDI agents to handle different types of uncertainty. However, it primarily is concerned with the interpretation of the BDI programs. Meanwhile, our approach not only supports the same uncertain modelling in [23] but also provides the safety of agent behaviours through formal verification under these uncertain beliefs.

Verifying BDI agents using model checking, via Java PathFinder [9], and theorem proving, using Isabelle/HOL [10] has also been explored. However, these use fixed schedulers for agent selections strategies, e.g. first-in-first-out for intention selection, and do not allow probabilistic action outcomes for the agents and uncertain beliefs. Verification and strategy synthesis have also been successfully applied to many traditional probabilistic systems (e.g. security systems or protocols) overviewed in [16]. The contribution of our work applied both verification and strategy synthesis to ensure correct and optimal BDI agent behaviours (which feature non-deterministic choices, probabilistic action outcomes, and uncertain beliefs) together with a fully automated tool.

## 9. Future work

There are many avenues for future work, some theoretical and some concerned with practical improvements to our tool.

Providing the capability to provide uncertain beliefs does not take away the inherent challenges of how specify these uncertainties in the first place. We have preliminary and ongoing, but promising, results that use the off-the-shelf computer vision algorithms, which can provide probability to pattern recognition, to aid the specification of these uncertain beliefs.

Once we accept the concepts of epistemic states, we can have a number of epistemic states rather than an epistemic state over the entire set of belief atoms. The agent could have different parts of its beliefs with different uncertainty natures or different revision rules. This can be for complexity purposes, too i.e. when certain subsets of beliefs do not influence each other, they can be kept in separate epistemic states to simplify the representation of an epistemic state by partitioning the beliefs.

Another line of future work is to synthesise strategy in our CAN-VERIFY tool. As we have mentioned in the reflection, the difficulty of providing such automated features is not in the technical implementation but in how to output the actual representation of strategies without involving the underlying bigraph syntax. One of the simplest strategies (which is also the one we focus on in this paper) are memoryless deterministic strategies, which pick a single action in each state for a MDP. In the BDI setting, the action would correspond to what semantics rules are applied, whereas the states the configuration of the agents. However, as the state of a BDI agent is encoded in the Bigraph language, to allow the users to know what configuration of the agents is in, we need to translate the bigraph to BDI configuration again, i.e. a decoder process. We anticipate such an automated feature would immensely support the control side of autonomous systems when required.

Finally, the current implementation of textual property specification in our tool relies on simple string matching, requiring exact wording and structure with limited flexibility in specifying user-defined properties. This approach allows to capture some common properties such as checking whether a belief will hold. However, it admittedly does not capture all possible properties (which could be specified in PCTL). Though translating generic natural language into PCTL formulas is outwith the scope of this work, we view this early-stage natural language support essential to make property specification more user-centric. For future work, we intend to support practitioners with formal methods training to directly provide formally specified property in PCTL. For other users, we plan to integrate with our tool an existing property elicitation interface such as NASA's Formal Requirements Elicitation Tool (FRET) [52] or large language models-based approaches e.g. [53] to support specification in less restricted natural language settings.

## 10. Conclusion

Quantitative verification and strategy synthesis is a powerful technique for analysing systems and synthesising strategies that exhibit non-deterministic, probabilistic behaviours for autonomous agents operating with uncertain beliefs.

We have shown how epistemic states can be employed to allow us to model uncertain beliefs within BDI agents, and how to model these agents in a quantitative verification setting. To achieve this, we translate the CAN language (which formalises the behaviour of a classical BDI agent), with added epistemic states to a Markov Decision Process model. This supports both non-deterministic decision-making (e.g. which plan to select), probabilistic agent action outcomes (e.g. imprecise actuators), and uncertain beliefs (e.g. noise sensors). The resulting model, $\text{CAN}_{+}^{m}$, is encoded and executed using Milner's bigraphs and the BigraphER tool. This allows quantitative analysis and strategy synthesis using popular probabilistic model-checking tools including PRISM and Storm. This approach is automated via the CAN-VERIFY, and aims to meet the growing demand 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.

## CRediT authorship contribution statement

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

## 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.

$$\frac{act : \psi \leftarrow \mu \quad \mu((l_i, m_i)) = p_i \quad \mathcal{W} \models \psi}{\langle \mathcal{W}, act \rangle \rightarrow_{p_i} \langle \mathcal{W} \circ (l_i, m_i), nil \rangle}\ act_p$$

$$\frac{\Delta = \{\varphi : P \mid (e' = \varphi \leftarrow P) \in \Pi \wedge e' = e\}}{\langle \mathcal{W}, e \rangle \rightarrow_1 \langle \mathcal{W}, e : (\!| \Delta |\!) \rangle}\ event$$

$$\frac{\langle n, \varphi : P \rangle \in \Delta \quad \mathcal{W} \models \varphi}{\langle \mathcal{W}, e : (\!| \Delta |\!) \rangle \rightarrow_1 \langle \mathcal{W}, P \rhd e : (\!| \Delta \setminus \{\langle n, \varphi : P \rangle\} |\!) \rangle}\ select(n)$$

$$\frac{\langle \mathcal{W}, P_1 \rangle \rightarrow_p \langle \mathcal{W}', P_1' \rangle}{\langle \mathcal{W}, P_1 \rhd P_2 \rangle \rightarrow_p \langle \mathcal{W}', P_1' \rhd P_2 \rangle}\rhd_; \quad \frac{}{\langle \mathcal{W}, (nil \rhd P_2) \rangle \rightarrow_1 \langle \mathcal{W}', nil \rangle}\rhd_\top$$

$$\frac{P_1 \neq nil \ \langle \mathcal{W}, P_1 \rangle \nrightarrow_1 \ \langle \mathcal{W}, P_2 \rangle \rightarrow_p \langle \mathcal{W}', P_2' \rangle}{\langle \mathcal{W}, P_1 \rhd P_2 \rangle \rightarrow_p \langle \mathcal{W}', P_2' \rangle}\rhd_\bot \quad \frac{\langle \mathcal{W}, P \rangle \rightarrow_p \langle \mathcal{W}', P' \rangle}{\langle \mathcal{W}, (nil; P) \rangle \rightarrow_p \langle \mathcal{W}', P' \rangle};_\top$$

$$\frac{\langle \mathcal{W}, P_1 \rangle \rightarrow_p \langle \mathcal{W}', P_1' \rangle}{\langle \mathcal{W}, (P_1; P_2) \rangle \rightarrow_p \langle \mathcal{W}', (P_1'; P_2) \rangle}; \quad \frac{\langle \mathcal{W}, P_1 \rangle \rightarrow_p \langle \mathcal{W}', P_1' \rangle}{\langle \mathcal{W}, (P_1 \| P_2) \rangle \rightarrow_p \langle \mathcal{W}', (P_1' \| P_2) \rangle}\|_1$$

$$\frac{\langle \mathcal{W}, P_2 \rangle \rightarrow_p \langle \mathcal{W}', P_2' \rangle}{\langle \mathcal{W}, (P_1 \| P_2) \rangle \rightarrow_p \langle \mathcal{W}', (P_1 \| P_2') \rangle}\|_2 \quad \frac{}{\langle \mathcal{W}, (nil \| nil) \rangle \rightarrow_1 \langle \mathcal{W}, nil \rangle}\|_\top$$

$$\frac{\mathcal{W} \models \varphi_s}{\langle \mathcal{W}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow_1 \langle \mathcal{W}, nil \rangle}G_s \quad \frac{\mathcal{W} \models \varphi_f}{\langle \mathcal{W}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow_1 \langle \mathcal{W}, ?false \rangle}G_f$$

$$\frac{P \neq P_1 \rhd P_2 \quad \mathcal{W} \not\models \varphi_s \quad \mathcal{W} \not\models \varphi_f}{\langle \mathcal{W}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow_1 \langle \mathcal{W}, goal(\varphi_s, P \rhd P, \varphi_f) \rangle}G_{init}$$

$$\frac{\mathcal{W} \not\models \varphi_s \quad \mathcal{W} \not\models \varphi_f \quad \langle \mathcal{W}, P_1 \rangle \rightarrow_p \langle \mathcal{W}', P_1' \rangle}{\langle \mathcal{W}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f) \rangle \rightarrow_p \langle \mathcal{W}', goal(\varphi_s, P_1' \rhd P_2, \varphi_f) \rangle}G_; \quad \frac{\mathcal{W} \not\models \varphi_s \quad \mathcal{W} \not\models \varphi_f \quad \langle \mathcal{W}, P_1 \rangle \nrightarrow_1}{\langle \mathcal{W}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f) \rangle \rightarrow_1 \langle \mathcal{W}, goal(\varphi_s, P_2 \rhd P_2, \varphi_f) \rangle}G_\rhd$$

**Fig. 12.** Intention-level CAN$_+$ semantics.

$$\frac{\langle n, e \rangle \in E^e}{\langle E^e, \mathcal{W}, \Gamma \rangle \Rightarrow_1 \langle E^e \setminus \{\langle n, e \rangle\}, \mathcal{W}, \Gamma \cup \{\langle n, e \rangle\} \rangle}A_{event}(n)$$

$$\frac{\langle n, P \rangle \in \Gamma \quad \langle \mathcal{W}, \langle n, P \rangle \rangle \rightarrow_p \langle \mathcal{W}', \langle n, P' \rangle \rangle}{\langle E^e, \mathcal{W}, \Gamma \rangle \Rightarrow_p \langle E^e, \mathcal{W}', (\Gamma \setminus \{\langle n, P \rangle\}) \cup \{\langle n, P' \rangle\} \rangle}A_{step}(n)$$

$$\frac{\langle n, P \rangle \in \Gamma \quad \langle \mathcal{W}, \langle n, P \rangle \rangle \nrightarrow_1}{\langle E^e, \mathcal{W}, \Gamma \rangle \Rightarrow_1 \langle E^e, \mathcal{W}, \Gamma \setminus \{\langle n, P \rangle\} \rangle}A_{update}(n)$$

**Fig. 13.** Agent-level CAN$_+$ semantics.

## Acknowledgement

This work is supported by the Engineering and Physical Sciences Research Council, under grant TransiT (EP/Z533221/1) and an Amazon Research Award on Automated Reasoning.

## Appendix A

We provide the full set of semantic rules of both intention-level semantics (Fig. 12) and agent-level semantics (Fig. 13) for CAN$_+$. As the full set of agent-level semantics is already explained in Section 4.3, we note that most of cases in intention-level semantics simply need to replace the belief base $\mathcal{B}$ with $\mathcal{W}$ and take into account the probability of other intention-level semantics, in particular, in construct of e.g. $P_1; P_2$.

## References

[1] M. Bratman, Intention, Plans, and Practical Reason, Harvard University Press, 1987.
[2] A.S. Rao, AgentSpeak (L): BDI agents speak out in a logical computable language, in: Proceedings of European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Springer, 1996, pp. 42–55.
[3] K.V. Hindriks, F.S.D. Boer, W.V.d. Hoek, J.-J.C. Meyer, Agent programming in 3APL, Auton. Agents Multi-Agent Syst. 2 (4) (1999) 357–401.
[4] M. Dastani, 2APL: a practical agent programming language, Auton. Agents Multi-Agent Syst. 16 (3) (2008) 214–248.
[5] R. Bordini, J. Hübner, M. Wooldridge, Programming Multi-Agent Systems in AgentSpeak Using Jason, vol. 8, John Wiley & Sons, 2007.
[6] S. Sardina, L.d. Silva, L. Padgham, Hierarchical planning in BDI agent programming languages: a formal approach, in: Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, 2006, pp. 1001–1008.
[7] M. Wooldridge, An Introduction to Multiagent Systems, John Wiley & Sons, 2009.
[8] M. Luckcuck, M. Farrell, L.A. Dennis, C. Dixon, M. Fisher, Formal specification and verification of autonomous robotic systems: a survey, ACM Comput. Surv. 52 (5) (2019) 1–41.
[9] L.A. Dennis, M. Fisher, M.P. Webster, R.H. Bordini, Model checking agent programming languages, Autom. Softw. Eng. 19 (1) (2012) 5–63.
[10] 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.
[11] H. Chen, Applications of cyber-physical system: a literature review, J. Ind. Integration Manag. 2 (03) (2017) 1750012.

[12] B. Archibald, M. Calder, M. Sevegnani, M. Xu, Probabilistic BDI agents: actions, plans, and intentions, in: Software Engineering and Formal Methods, Springer International Publishing, 2021, pp. 262–281.
[13] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Elsevier, 2004.
[14] H. Geffner, B. Bonet, A concise introduction to models and methods for automated planning, Synth. Lect. Artif. Intell. Mach. Learn. 8 (1) (2013) 1–141.
[15] Y. Abdeddaı, E. Asarin, O. Maler, et al., Scheduling with timed automata, Theor. Comput. Sci. 354 (2) (2006) 272–300.
[16] M. Kwiatkowska, D. Parker, Automated verification and strategy synthesis for probabilistic systems, in: Automated Technology for Verification and Analysis, Springer, 2013, pp. 5–22.
[17] M. Winikoff, L. Padgham, J. Harland, J. Thangarajah, Declarative and procedural goals in intelligent agent systems, in: The 8th International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufman, 2002.
[18] S. Sardina, L. Padgham, A BDI agent programming language with failure handling, declarative goals, and planning, Auton. Agents Multi-Agent Syst. (2011) 18–70.
[19] R. Milner, The Space and Motion of Communicating Agents, Cambridge University Press, 2009.
[20] B. Archibald, M. Calder, M. Sevegnani, Probabilistic bigraphs, Form. Asp. Comput. 34 (2) (2022) 1–27.
[21] R. Bellman, A Markovian decision process, J. Math. Mech. (1957) 679–684.
[22] J. Ma, W. Liu, A framework for managing uncertain inputs: an axiomization of rewarding, Int. J. Approx. Reason. 52 (7) (2011) 917–934 [Online]. Available: https://doi.org/10.1016/j.ijar.2011.05.004.
[23] K. Bauters, K. McAreavey, W. Liu, J. Hong, L. Godo, C. Sierra, Managing different sources of uncertainty in a bdi framework in a principled way with tractable fragments, J. Artif. Intell. Res. 58 (2017) 731–775.
[24] M. Sevegnani, M. Calder, BigraphER: rewriting and analysis engine for bigraphs, in: The 28th International Conference on Computer Aided Verification, 2016, pp. 494–501 [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_27.
[25] M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: verification of probabilistic real-time systems, in: The 23rd International Conference on Computer Aided Verification, in: LNCS, vol. 6806, Springer, 2011, pp. 585–591.
[26] B. Archibald, M. Calder, M. Sevegnani, M. Xu, Quantitative verification and strategy synthesis for BDI agents, in: Proceedings of NASA Formal Methods, Springer Nature Switzerland, 2023, pp. 241–259.
[27] 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.
[28] A. Di Pierro, H. Wiklicky, An operational semantics for probabilistic concurrent constraint programming, in: The 1998 International Conference on Computer Languages, IEEE, 1998, pp. 174–183.
[29] B. Archibald, C. Muffy, M. Sevegnani, Conditional bigraphs, in: International Conference on Graph Transformation, Springer, 2020, pp. 3–19.
[30] H. Hansson, B. Jonsson, A logic for reasoning about time and reliability, Form. Asp. Comput. 6 (5) (1994) 512–535.
[31] J. Päßler, M.H. ter Beek, F. Damiani, S.L. Tapia Tarifa, E.B. Johnsen, Formal modelling and analysis of a self-adaptive robotic system, in: International Conference on Integrated Formal Methods, Springer, 2023, pp. 343–363.
[32] G.R. Silva, J. Päßler, J. Zwanepol, E. Alberts, S.L.T. Tarifa, I. Gerostathopoulos, E.B. Johnsen, C.H. Corbato, Suave: an exemplar for self-adaptive underwater vehicles, in: 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE, 2023, pp. 181–187.
[33] 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.
[34] S. Benford, M. Calder, T. Rodden, M. Sevegnani, On lions, impala, and bigraphs: modelling interactions in physical/virtual spaces, ACM Trans. Comput.-Hum. Interact. 23 (2) (2016) 1–56.
[35] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Proceedings of Workshop on Logic of Programs, Springer, 1981, pp. 52–71.
[36] D. Remenska, Bringing model checking closer to practical software engineering, 2016.
[37] M.Y. Vardi, On the complexity of epistemic reasoning, IBM Thomas J. Watson Research Division, 1989.
[38] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 2018.
[39] A. Hartmanns, M. Klauck, The modest state of learning, sampling, and verifying strategies, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2022, pp. 406–432.
[40] F.A. Oliehoek, C. Amato, et al., A Concise Introduction to Decentralized POMDPs, vol. 1, Springer, 2016.
[41] C. Novakovic, D. Parker, Automated formal analysis of side-channel attacks on probabilistic systems, in: Proc. 24th European Symposium on Research in Computer Security (ESORICS'19), in: LNCS, vol. 11735, Springer, 2019, pp. 319–337.
[42] D. Silver, J. Veness, Monte-Carlo planning in large pomdps, Adv. Neural Inf. Process. Syst. 23 (2010).
[43] G. Norman, D. Parker, X. Zou, Verification and control of partially observable probabilistic systems, Real-Time Syst. 53 (2017) 354–402.
[44] L. Padgham, D. Singh, Situational preferences for BDI plans, in: The 2013 International Conference on Autonomous Agents and Multi-Agent Systems, 2013, pp. 1013–1020.
[45] F. Meneguzzi, L. De Silva, Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning, Knowl. Eng. Rev. 30 (1) (2015) 1–44.
[46] L. De Silva, F.R. Meneguzzi, B. Logan, BDI agent architectures: a survey, in: The 29th International Joint Conference on Artificial Intelligence, 2020.
[47] R.H. Bordini, A.L.C. Bazzan, R.D.O. Jannone, D.M. Basso, R.M. Vicari, V.R. Lesser, AgentSpeak (XL) efficient intention selection in BDI agents via decision-theoretic task scheduling, in: The First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 3, 2002, pp. 1294–1302.
[48] M. Xu, K. McAreavey, K. Bauters, W. Liu, Intention interleaving via classical replanning, in: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence, IEEE, 2019, pp. 85–92.
[49] M. Xu, K. Bauters, K. McAreavey, W. Liu, A formal approach to embedding first-principles planning in BDI agent systems, in: International Conference on Scalable Uncertainty Management, Springer, 2018, pp. 333–347.
[50] B. Logan, J. Thangarajah, N. Yorke-Smith, Progressing intention progression: a call for a goal-plan tree contest, in: Proceedings of International Conference on Autonomous Agents and Multiagent Systems, 2017, pp. 768–772.
[51] Homepage, Intention progression competition, https://www.intentionprogression.org/.
[52] 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, REFSQ 2022, Birmingham, UK, March 21–24, 2022, Proceedings, Springer, 2022, pp. 96–111.
[53] M. Cosler, C. Hahn, D. Mendoza, F. Schmitt, C. Trippel, nl2spec: interactively translating unstructured natural language to temporal logics with large language models, in: International Conference on Computer Aided Verification, Springer, 2023, pp. 383–396.