Quantitative Verification and Strategy Synthesis for BDI Agents

Blair Archibald, Muffy Calder, Michele Sevegnani, and Mengwei Xu

University of Glasgow, Glasgow, UK,
{`blair.archibald, muffy.calder, michele.sevegnani, mengwei.xu`}`@glasgow.ac.uk`

Abstract. Belief-Desire-Intention (BDI) agents feature probabilistic outcomes, *e.g.* the chance an agent tries but fails to open a door, and non-deterministic choices: what plan/intention to execute next? We want to reason about agents under both probabilities and non-determinism to determine, for example, probabilities of mission success and the *strategies* used to maximise this. We define a Markov Decision Process describing the semantics of the Conceptual Agent Notation (Can) agent language that supports non-deterministic event, plan, and intention selection, as well as probabilistic action outcomes. The model is derived through an encoding to Milner's Bigraphs and executed using the BigraphER tool. We show, using probabilistic model checkers PRISM and Storm, how to reason about agents including: probabilistic and reward-based properties, strategy synthesis, and multi-objective analysis. This analysis provides verification and optimisation of BDI agent design and implementation.

Keywords: BDI Agents; Quantitative Verification; Strategy Synthesis; Markov Decision Process; Bigraphs; PRISM; Storm

1 Introduction

BDI agents [1] are a popular architecture for developing rational agents where (B)eliefs represent what an agent knows, (D)esires what the agent wants to bring about, and (I)ntentions the desires the agent is currently acting on. BDI agents have inspired many agent programming languages including AgentSpeak [2], Can [3], 3APL [4], and 2APL [5] along with a collection of mature software including JACK [6], Jason [7], and Jadex [8].

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 [9]: 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 on which plan is selected from a set of possible plans (plan selection).

The deployment of BDI-based systems raises concerns of trustworthiness. For example, 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 execution of one intention can make it impossible to progress another. 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. Furthermore, the outcome of an action may be 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 [10]. Given the number of 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.

Verifying BDI agent behaviours through model checking and theorem proving has been well explored. For example, the authors apply the Java PathFinder model-checker (resps. Isabelle/HOL proof assistant) to verify BDI programs in the work [11] (resp. [12]). Unfortunately, they do not adequately represent agent behaviours in cyber-physical robotics systems (*e.g.* surveyed in [13]) with imprecise actuators. To reason with the quantitative behaviours of BDI agents, the authors of [14] 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 [15, 16].

We show how to combine and apply quantitative verification and strategy synthesis [17, 18] within BDI agents allowing us to both determine, *e.g.* the probability an agent successfully completes a mission under environmental uncertainty, and also a method to resolve the non-determinism required for intention/event/plan selection. We focus on the CAN language [3, 19] which features a high-level agent programming language that captures the essence of BDI concepts without describing implementation details such as data structures. As a superset of most well-known AgentSpeak [2], CAN includes advanced BDI agent behaviours such as reasoning with *declarative goals* and *failure recovery*, which are necessary for our examples discussed in Section 4. Importantly, 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 our previous work [14] developing an executable *probabilistic* semantics of CAN [3], based on Milner's Bigraphs [20]. Specifically, we use probabilistic bigraphs [21], that assigns probabilities to transitions (graph rewrites). Previously, we used manually-crafted strategies (*e.g.* fixed schedule) to resolve non-determinism. Instead, we keep these selections as non-deterministic choices and encode them using action bigraphs [21] (which supports modelling non-deterministic actions). This provides a model of Can based on a Markov decision process (MDP) [22], that we denote as $\textsc{Can}^m$. 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, imprecise actuator. For analysis, we export, using BigraphER [23], the underlying MDP to the popular probabilistic model checkers PRISM [24] and Storm [25]. This includes probabilistic and reward-based properties, strategy synthesis, and multi-objective analysis. In particular, temporal logics provide an expressive means of formally specifying the requirement properties when synthesising strategies that are guaranteed to be correct (at least with respect to the specified model and properties)

We make the following research contributions:

- an MDP model of the Can semantics, supporting non-deterministic selections and probabilistic action outcomes;
- an executable MDP model of Can with BigraphER for quantitative verification and (optimal) strategy synthesis through PRISM and Storm;
- a simple example of smart manufacturing computes the probability analysis of mission success and strategy synthesis, and a simple example of a rover computes the reward probability of mission success and strategy synthesis.

Outline In Section 2 we recall BDI agents and an MDP. In Section 3 we propose our approach. In Section 4 we evaluate our approach to smart manufacturing and rover examples. We discuss related work in Section 5 and conclude in Section 6.

2 Background

2.1 CAN

Can language formalises a classical BDI agent consisting of a belief base $\mathcal{B}$ and a plan library Π. The belief base $\mathcal{B}$ is a set of formulas encoding the current beliefs and has belief operators for entailment (*i.e.* $\mathcal{B} \models \varphi$), and belief atom addition (resp. deletion) $\mathcal{B} \cup \{b\}$ (resp. $\mathcal{B} \setminus \{b\}$)[1]. A plan library Π is a collection of plans of the form $e : \varphi \leftarrow P$ with e the triggering event, φ the context condition, and P the plan-body. The triggering event e specifies why the plan is relevant, while the context condition φ determines *when* the plan-body P is applicable.

The Can semantics are specified by two types of transitions. The first, denoted $\rightarrow$, specifies *intention-level* evolution on intention-level 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 agent-level configurations $\langle E^e, \mathcal{B}, \Gamma \rangle$, detailing how to execute a complete agent where E^e is the set of pending external events to address (desires) and Γ a set of partially executed plan-bodies (intentions). The intention-level Can configurations $\langle \mathcal{B}, P \rangle$ can be seen a special case of $\langle E^e, \mathcal{B}, \Gamma \rangle$ where E^e is an arbitrary set of event and $P \in \Gamma$. We denote configurations as $\mathcal{C}$.

[1] Any logic is allowed providing entailment is supported.

$$\frac{act : \psi \leftarrow \langle \phi^-, \phi^+ \rangle \quad \mathcal{B} \vDash \psi}{\langle \mathcal{B}, act \rangle \rightarrow \langle (\mathcal{B} \setminus \phi^- \cup \phi^+), nil \rangle}\ act \qquad \frac{\varphi : P \in \Delta \quad \mathcal{B} \models \varphi}{\langle \mathcal{B}, e : (\mid \Delta \mid) \rangle \rightarrow \langle \mathcal{B}, P \rhd e : (\mid \Delta \setminus \{\varphi : P\} \mid) \rangle}\ select$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1' \| P_2) \rangle}\ \|_1 \qquad \frac{\langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1 \| P_2') \rangle}\ \|_2$$

Fig. 1. Examples of intention-level CAN semantics.

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

$$\frac{P \in \Gamma \quad \langle \mathcal{B}, P \rangle \nrightarrow}{\langle E^e, \mathcal{B}, \Gamma \rangle \Rightarrow \langle E^e, \mathcal{B}, \Gamma \setminus \{P\} \rangle}\ A_{update}$$

Fig. 2. Agent-level CAN semantics.

Fig. 1 gives some semantics rules for evolving an intention. For example, *act* handles the execution of an action (in the form of $act = \psi \leftarrow \langle \phi^-, \phi^+ \rangle$), when the pre-condition ψ is met, resulting in a belief state update ($\mathcal{B} \setminus \phi^- \cup \phi^+$). Rule *select* chooses an applicable plan from a set of relevant plans (*i.e.* $\mathcal{B} \models \varphi$ and $\varphi : P \in \Delta$) while retaining un-selected plans as backups (*i.e.* $P \rhd e : (\mid \Delta \setminus \{\varphi : P\} \mid)$). Rules $\|_1$ and $\|_2$ specify how to execute (interleaved) concurrent programs (within an intention). The full intention-level semantics is given in Appendix A. The agent-level semantics are given in Fig. 2. The rule A_{event} handles external events, that originate from the environment, by adopting them as intentions. Rule A_{step} selects an intention and evolves a single step w.r.t. the intention-level transition, while A_{update} discards unprogressable intentions (either succeeded, or failed).

2.2 Markov Decision Processes

A Markov decision process (MDP) [22] is a tuple $\mathcal{M} = (S, \bar{s}, \alpha, \delta)$ where S is a set of states, $\bar{s}$ an initial state, α 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 An MDP Model of CAN Semantics

MDPs model systems with nondeterministic and probabilistic behaviour. To use an MDP with the Can semantics 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 $\langle E^e, \mathcal{B}, \Gamma\rangle$ of Can. The state space is $S \subseteq 2^{E^e} \times 2^{\mathcal{B}} \times 2^{\Gamma}$ where the exact subset of states is determined by the specific program we are modelling[2]. An initial state of a Can^m is $\bar{s} = \langle E^e_0, \mathcal{B}_0, \Gamma_0\rangle$. In practice, including our examples in Section 4, this usually has the form $E^e_0 = \{e_1, \cdots, e_j\}$ (a set of tasks), $\mathcal{B}_0 = \{b_1, \cdots, b_k\}$ (an initial set of beliefs, *e.g.* about the environment), $\Gamma_0 = \emptyset$ (no intentions yet), and $j, k \in \mathbb{N}^+$.

The Can semantics are defined using operational semantics with transitions over configurations $\mathcal{C} \to \mathcal{C}'$ (see Section 2.1). As we reason with probabilistic action outcomes of agents, we instead use probabilistic transitions $\mathcal{C} \to_p \mathcal{C}'$, *i.e.* this transition happens with probability p [26]. In our case, probabilities are introduced by uncertain action outcomes of the agents (see Section 3.1).

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 δ such that Eqs. (2) and (3) hold:

$$\frac{\lambda_1 \quad \lambda_2 \quad \cdots \quad \lambda_n}{\mathcal{C} \to_p \mathcal{C}'}\; rule \tag{1}$$

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

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

Condition (2) says a transition of Can^m is only enabled if the transition would be enabled in Can, *i.e.* the premises λ_i of *rule* are all met. Condition (2) defines the probability of transitioning from $\mathcal{C}$ to $\mathcal{C}'$ in Can^m as the same as the probability of transitioning in Can. 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. 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.

3.1 Probabilistic Action Outcomes

Probabilistic transitions occur when we add support for probabilistic action outcomes for agents. In Can, the semantic rule *act* gives a fixed outcome (belief changes in the semantics; but also environment changes in real application) when an agent action is executed. 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

[2] We determine this by symbolically executing the program as we convert to an MDP.

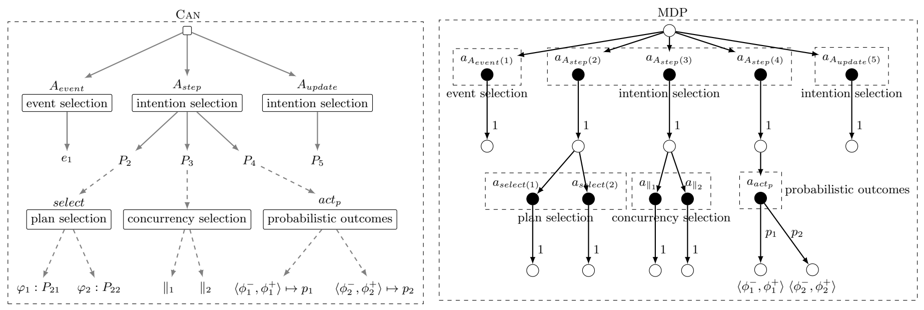


Fig. 3. Left: CAN semantic rule possibilities highlighting event, intention, plan, and concurrency selection, and probabilistic agent action outcomes. Solid lines are agent-level transitions and dashed lines are intention-level. Right: Corresponding MDP model of CAN semantic rules with empty circles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

outcomes in agent actions, we introduce a new *probabilistic* semantic rule (same as in [14]) act_p where $\mu = [(\phi_1^-, \phi_1^+) \mapsto p_1, \ldots, (\phi_n^-, \phi_n^+) \mapsto p_n]$ is a user-specified outcome distribution where $\mu(\phi_i^-, \phi_i^+) = p_i$ and $\sum_{i=1}^{n} p_i = 1$.

$$\frac{act : \psi \leftarrow \mu \quad \mu(\phi_i^-, \phi_i^+) = p_i \quad \mathcal{B} \vDash \psi}{\langle \mathcal{B}, act \rangle \rightarrow_{p_i} \langle (\mathcal{B} \setminus \phi_i^- \cup \phi_i^+), nil \rangle}\ act_p$$

For mapping intention-level CAN configurations to MDP states we use the fact that $\langle \mathcal{B}, P \rangle$ is a special case of $\langle E^e, \mathcal{B}, \Gamma \rangle$ where E^e is an arbitrary set of event 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}:

$$\delta(\mathcal{C}, a_{act_p})(\mathcal{C}') = p_i \text{ s.t. } \mathcal{C} = \langle \mathcal{B}, act \rangle,\ act : \psi \leftarrow \mu,\ \mathcal{B} \vDash \psi,$$
$$\mu(\phi_i^-, \phi_i^+) = p_i, \text{ and } \mathcal{C}' = \langle \mathcal{B} \setminus \phi_i^- \cup \phi_i^+, nil \rangle$$

3.2 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 a parameterised rules, *e.g.* *select*(n) as follows:

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

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.

3.3 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 3.2, 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:

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

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

$$\frac{\langle n, P\rangle \in \Gamma \quad \langle \mathcal{B}, \langle n, P\rangle\rangle \not\rightarrow_1}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow_1 \langle E^e, \mathcal{B}, \Gamma \setminus \{\langle n, P\rangle\}\rangle} A_{update}(n)$$

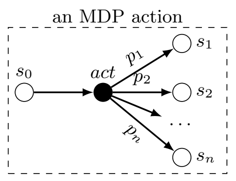
Event parameters are specified by numbering them based on an ordering on the full set of events, e.g. $\langle n, e\rangle$ with $n \in \mathbb{N}^+$ as an identifier. We identify (partially executed) intentions based on the identifier of the top level plan that led to this intention, *e.g.* for $P \in \Gamma$ 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. Formally, we have:

$$\delta(\langle E^e, \mathcal{B}, \Gamma\rangle, a_{A_{step}(n)})(\langle E^e, \mathcal{B}', \Gamma \setminus \{\langle n, P\rangle\}) \cup \{\langle n, P'\rangle\}\rangle) = p \text{ iff}$$
$$\langle n, P\rangle \in \Gamma \text{ and } \delta(\langle \mathcal{B}, \langle n, P\rangle\rangle, a_{rule})(\langle \mathcal{B}', \langle n, P'\rangle\rangle) = p$$

where a_{rule} denotes the MDP action for the equivalent semantic rule in Can that handles the intention-level transition of $\langle \mathcal{B}, \langle n, P\rangle\rangle \rightarrow_p \langle \mathcal{B}', \langle n, P'\rangle\rangle$.

3.4 Rewards

While an MDP allows action rewards to be assigned to any action, and therefore Can rule, they are particularly useful for the parameterised rules. In practice, as it is difficult to specify all current states of an agent (needed for the configuration), we apply rewards based only on the action chosen, *e.g.* $R(\mathcal{C}, a_{rule}) = R(a_{rule}) = r_{rule}$. With this, we can choose preferred parameters by assigning higher reward values, *e.g.* $R(a_{select(1)}) < R(a_{select(2)})$. Usually we give non-zero rewards to MDP actions that correspond to selection (*e.g.* plan selection) for strategy synthesis later on. For other MDP actions the reward is 0.



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 $\frac{w_i}{\sum_{j=1}^{n} w_j} = p_i$

Fig. 4. 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*.

4 Implementation and Examples

Using a simple smart manufacturing and rover example, we show how our approach can quantitatively analyse/verify agent programs and synthesise the (optimal) strategies. Specifically, we evaluate the probabilistic properties for smart manufacturing and reward-based properties for the rover example together with their optimal strategy synthesis. The results show we can detect undesired executions (that result in mission failure) and generate different optimal strategies that can maximise either probability-based or reward-based objective. While we only give details of two simple cases, users of the executable model can "run" models with different external events and plan libraries. The examples shown in this paper and instructions on reproducibility are open available in [27].

4.1 Bigraph Encoding of CAN^m Model

We use Milner's bigraphs [20]—a graph-based rewriting formalism—to encode our CAN^m model. As a graph-based rewriting formalism, over customised rules called *reaction rules*, bigraphs provide an intuitive diagrammatic representation to model the execution process of the systems. Applying a reaction rule, $L \longrightarrow R$, replaces an occurrence of bigraph L (in a bigraph) with bigraph R. Given an initial bigraph (*i.e.* initial system state) and a set of reaction rules (*i.e.* system dynamics), we obtain a transition system capturing system behaviours for formal verification. 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 $\mathtt{r}'$. 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'. Figure 4 depicts how to encode any MDP action in bigraphs. To execute our bigraph model, we employ BigraphER [23], an open-source language and toolkit for bigraphs. It allows exporting 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 are exported for quantitative analysis and strategy synthesis in PRISM and Storm. We use PRISM[3] (for

[3] PRISM currently does not support reward import.

non-reward properties) and Storm (for reward-based properties) 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) [28]. $\mathcal{P}_{max=?}\mathbf{F}[\phi]$ expresses the maximum probability of ϕ holding *eventually* in all possible resolutions of non-determinism.

4.2 Example: Smart Manufacturing

We revisit the robotic packaging scenario from [14] where a robot packs products and moves them to a storage area. Previously, this example was quantitatively analysed using probabilistic model checking, but all non-determinism was resolved using pre-defined strategies (fixed, round-robin, probabilistic choice). Here, we wish to *find* a good strategy without assuming one.

The example is as follows: a robot is designed to pick a product from a production line, insulate them with either cheap or expensive wrapping bags (to prevent decay) and then move them to storage. Complexity arises from: (1) success depends on *when* a product is packed (*e.g.* before it decays), (2) *when* a product is packed determines which wrappings are applicable as earlier packing means cheaper bags, and (3) both wrappings introduce uncertainty as they may fail to insulate or break.

The agent program for a scenario with two initial products is given in Listing 1.1. We assume the agent uses a propositional logic with numerical comparisons. Products awaiting processing are captured by external events in line 4. The agent responds to the events using a declarative goal on line 6 stating it wants to achieve the state success1 (*i.e.* wrapped and moved) through addressing the (internal) event process_product1; failing if failure1 (*i.e.* dropped or decayed) ever becomes true. Two plans (in lines 7-8), representing the different wrappings, handle the event process_product1 depending on the deadline for the product. Event product2 is handled similarly (line 9–11). We encode (discrete) temporal information for the deadline as agent belief atoms. This should not be viewed as general support clocks in an MDP. Instead, these temporal information is simply modelled as numerical belief atoms and we update these belief atoms in the background, without executing any explicit MDP action. The deadline decreases after a step of *any* intention or the selection of *any* event. We have $deadline_1 = 10$ and $deadline_2 = 14$ as initial deadlines of product1 and product2 in line 2. The choice of these initial values was made by the agent designer. Our approach enables the analysis of alternative values quantitatively before deploying the agent. There is a probabilistic outcome for the agent action of both wrap_standard1 and move_product_standard1, such that they carry a 30% chance of causing the belief failure1 to hold by failing to insulate and dropping the product accidentally. Meanwhile, there is only 10% change of causing insulation failure or product dropping by action wrap_premium1 and move_product_premium1. Due to space limits, we omit the action descriptions. Full agent examples are online [27].

Quantitative Verification and Strategy Synthesis. For analysis we label states where properties of interest hold. We use PS1 and PS2 to denote product1

Listing 1.1. Can agent for smart manufacturing

```
// Initial belief bases
deadline₁ = 10, deadline₂ = 14
// External events
product1, product2
// Plan library
product1 : true <- goal(success1,process_product1,failure1).
process_product1 : deadline₁ ≥ 3 <- wrap_standard1; move_product_standard1.
process_product1 : deadline₁ ≥ 0 <- wrap_premium1; move_product_premium1.
product2 : true <- goal(success2,process_product2,failure2).
process_product2 : deadline₂ ≥ 3 <- wrap_standard2; move_product_standard2.
process_product2 : deadline₁ ≥ 3 <- wrap_premium2; move_product_premium2.
```

Listing 1.2. A list of properties with its associated value for smart manufacturing where PS1 and PS2 denote product1 and product2 successfully being processed, and Pch1 and Pch2 denote cheap bag selected for product1 and product2, respectively.

```
𝒫min=?F[PS1 ∧ PS2] (value 0)
𝒫max=?F[PS1 ∧ PS2] (value 0.6561)
𝒫max=?F[PS1 ∧ PS2 ∧ Pch1 ∧ Pch2] (value 0)
𝒫max=?F[PS1 ∧ PS2 ∧ (Pch1 ∨ Pch2)] (value 0.3969)
```

and product2 being successfully processed by the robot. Pch1 and Pch2 hold when the cheaper bag was selected to handle product1 and product2 respectively. A full list of properties checked for this example is in Listing 1.2.

Property $\mathcal{P}_{min=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2}]$ checks the minimum probability of both products 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 handle both products, *e.g.* careless decision making causes failed deadlines. Property $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2}]$ determines the best possible outcome (both products processed) and returns a value of 0.6561[4], which implies there exists an adversary that the robot can handle both products with moderate success. Given this property, PRISM can automatically synthesise an adversary (strategy) for achieving this property. That is, a list of MDP actions to be taken in each state. Here the optimal adversary instructs the robot to wrap more urgent products (*i.e.* product1) first until it is packed and then switch to wrap the other product. As expected, in both cases the expensive bag is used. Only after both are wrapped does the robot move them to storage.

The property $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2} \wedge \mathsf{Pch1} \wedge \mathsf{Pch2}]$ checks if there is a way to successfully handle both products while using cheap bags for both of them. The value is 0, confirming it is impossible to do so. We can use the property $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2} \wedge (\mathsf{Pch1} \vee \mathsf{Pch2})]$ to determine if it is possible to use a cheap bag for either product. This is possible ($p = 0.3969$) by adapting the optimal strategies from before to use a cheap bag for product1.

[4] This probability is never 1 as there is always a chance bags fail regardless of type.

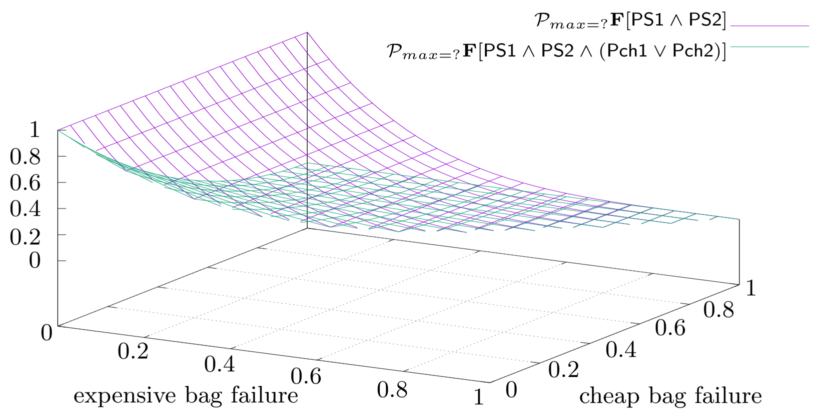


Fig. 5. Value of the property $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2}]$ and $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2} \wedge (\mathsf{Pch1} \vee \mathsf{Pch2})]$ with increasing failure probability in cheap and expensive bags.

Action Outcome Analysis. The effects of different failure probability for cheap and expensive bag are shown in Fig. 5 where the probability of bag failing to insulate or breaking is increased from 0 to 1. We can see that the value of the property $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2}]$ and $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2} \wedge (\mathsf{Pch1} \vee \mathsf{Pch2})]$ shows a decreasing trend with increasing failure probability in both cheap and expensive bag. When the failure probability of both types of bags equals to 0 or 1, the values of these two properties coincide with each other with either total success of probability 1 or total failure of probability 0. As expected, the probability of successfully handling two products is always higher than the one which requires cheap bags to be used because of the larger failure probability from the cheap bag than the expensive bag.

4.3 Example: Rover

We consider a rover scenario where the rover travels to a set of sites assigned by the mission centre for scientific experiments, *e.g.* to collect rocks or analyse soil. Given multiple sites to visit, the rover must *choose* one. Once chosen, the robot must then decide the route to use: some routes are shorter than others (plan selection). The mission is to successfully visit all sites, perform required experiments, and return to base.

Listing 1.3. CAN agent for rover.

```
1  // Initial belief bases
2  at_base
3  // External events
4  site1 , site2
5  // Plan library
6  site1 : true <- experiment_site1;return_to_base.
7  site2 : true <- experiment_site2;return_to_base.
8  experiment_site1 : at_base <- move_base_to_site1; perform_experiment_site1.
9  experiment_site1 : at_site1 <- perform_experiment_site1.
10 experiment_site1 : at_site2 <- move_site2_to_site1; perform_experiment_site1.
11 experiment_site2 : at_base <- move_base_to_site2; perform_experiment_site2.
12 experiment_site2 : at_site1 <- move_site1_to_site2; perform_experiment_site1.
13 experiment_site2 : at_site2 <- perform_experiment_site2.
14 return_to_base: at_base <- do_nothing.
15 return_to_base: at_site1 <- move_site1_to_base.
16 return_to_base: at_site2 <- move_site2_to_base.
```

To illustrate how much impact careless interleavings can make to the resulting agent behaviours, we use a very simplified scenario with only two sites to visit (*i.e.* a very small plan library). The agent program is in Listing 1.3. The rover has two sites to visit, which are captured by external events site1 and site2 in line 4, and is initially at the base (at_base in line 2). To address event site1, the plan on line 6 instructs the rover to pursue two ordered (internal) events, namely experiment_site1 and return_to_base. The first event experiment_site1 can be achieved by plans from lines 8 to 10 depending on where the rover is. For example, if the rover is at the base, the plan on line 8 instructs it to move to site 1 and perform necessary experiments. After successfully performing experiments the rover returns to base (return_to_base) through plans in lines 14–16. Event site2 can be handled in a similar way. In this case, we assume each moving agent action (*e.g.* move_base_to_site1) always succeeds. It allows us to easily reason about reward-based properties as any state with the reachability probability of less than 1 will always give an infinite reward. Full agent examples including action descriptions are online [27]. We use SS1 and SS2 to denote site1 and site2 successfully being processed by the rover and use Storm for model analysis.

We first check property $\mathcal{P}_{min=?}\mathbf{F}[\mathsf{SS1} \wedge \mathsf{SS2}]$ to see if there is a case where neither site visit is successful and, unexpectedly, the return value of 0 confirms this is possible. This shows, even in such a simple case, careless interleaving can cause issues: in this case movements back and forth between different locations without processing. For example, the rover may have moved to site 1, but before performing the experiment at site 1, it decides to address event site2 and moves to site 2. This behaviour then repeats in reverse. As the CAN semantics (semantic rule $\rhd_\perp$ in Appendix A) remove used plans on failure, the rover can enter a situation where there is no plan left to move and the mission cannot continue. We then check the property $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{SS1}\wedge\mathsf{SS2}]$ whose value is 1, confirming there is a way to analyse both sites. However, the optimal adversary (regarding the probability) returned by Storm makes unnecessary, but non-detrimental, movement between locations. To ensure the rover achieves the tasks while minimising trav-

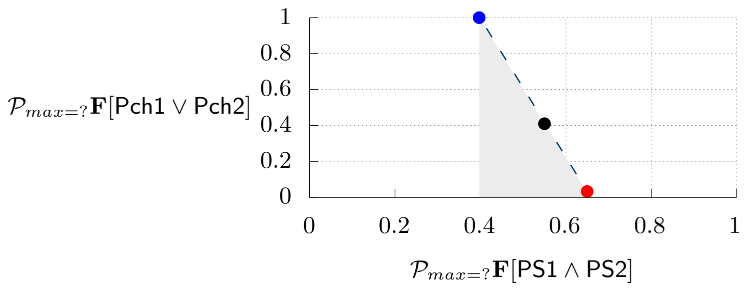


Fig. 6. Trade-offs between property $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{PS1} \wedge \mathsf{PS2}]$ (successfully handling two products) and $\mathcal{P}_{max=?}\mathbf{F}[\mathsf{Pch1} \vee \mathsf{Pch2}]$ (using cheap bag for either product). The red and blue point stand for two possible deterministic adversaries that always make the same choice in a given state of the model. Any point (*e.g.* black one) in the line between red and blue point represents a pair of mission objectives having randomised adversaries.

elling distance, we use rewards properties $\mathcal{R}_{min=?}\mathbf{F}[\mathsf{SS1} \wedge \mathsf{SS2}]$ and the expected strategy (visit and process each in turn and then return) is synthesised.

4.4 Discussion

Our framework gives agent designers an indication of the type of strategies that may be needed for a given application. For example, it gives confidence to either use a fixed strategy, such as the ordered schedule in the rover example, or justify the need for advanced planning capability. As we target the *semantics* rather than a specific implementation, it is possible to modify these in future to determine if other languages might be more suitable before implementation. For instance, in the rover example, the decision by CAN to throw away failed plans caused issue, while a different language design could avoid this pitfall.

The framework also allows verifying several, possibly conflicting, quantitative properties of an agent system. For example, we can ask how to maximise the probability of achieving the packing tasks while using cheap bags. PRISM can compute (approximately) the Pareto curve [29] shown in Fig. 6, which provides a useful visualization of trade-offs between different mission objectives and can help the agent designers prioritize objectives. Once the agent designer selects a combination of mission objective values in the line, a corresponding strategy can be automatically synthesized. In detail, the blue and red points stand for two pairs of objective values that have deterministic adversaries *i.e.* they always make the same choice in a state of the model. Any point (*e.g.* the black dot) on the line (except blue and red points) represents the pair of objective values that can be achieved by randomised strategy that makes an initial one-off random non-deterministic choice. However, it remains unclear how to interpret these randomised strategies, and this is an interesting area for future work.

We also note it is difficult to reason about the accumulated rewards for reaching some target set of states if these states cannot be eventually reached with probability 1. A good example would be to maximise the probability of achieving a mission state while minimising the cost of reaching it. 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). 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.

5 Related Work

Optimal decision-making under uncertainty is a core problem in Artificial Intelligence (AI). A prime example is planning [15, 16]: 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) [30]. 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 [31].

The BDI community is interested in event, plan and intention selection strategies and 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 overwhelming non-determinism through ordering—either statically [7] or at run-time [32]—to enforce simple deterministic behaviours. While desirable to exploit the highest ordered choices, it may be worthwhile exploring other non-highest order ones every now and then to avoid being stuck in a local maximum. Some selection strategies use advanced planning algorithms [33, 34]. For example, in [35] 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 [36]. An increasingly popular topic is intention progression [37], *e.g.* the contest [38], that helps the agent to make better decisions on event/plan/intention selection. 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. Finally, it is not a new idea to integrate advanced decision-making techniques into BDI. There is a large body of work [33] to employ planning to synthesise new plans to achieve an event when no pre-defined plan worked or exists. For example, work [39] shows how the integration of planning and BDI can be done at the semantic level.

Verifying BDI agents using model checking, via Java PathFinder [11], and theorem proving, using Isabelle/HOL [12] 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. Verification and strategy synthesis have also been successfully applied to many traditional probabilistic systems (*e.g.* security systems or protocols) overviewed in [18]. The contribution of our work applied both verification and strategy synthesis to ensure correct and optimal BDI agent behaviours (which features non-deterministic choices and probabilistic action outcomes) with the potentiality such as for multi-objective analysis.

6 Conclusions

Quantitative verification is a powerful technique for analysing systems that exhibits non-deterministic and probabilistic behaviours, allowing us to verify and synthesise strategies for autonomous agents operating in uncertain environments.

We have translated the Can language, which formalises the behaviour of a classical BDI agent, to an Markov Decision Process model. This supports both non-deterministic decision-making (*e.g.* which plan to select) and probabilistic agent action outcomes (*e.g.* imprecise actuators). The resulting model, Can^m, is encoded and executed using Milners bigraphs and the BigraphER tool. This allows quantitative analysis and strategy synthesis using popular probabilistic model checking tools including PRISM and Storm.

Through two simple examples, we have shown our approach can help the agent developers to reason about probability and reward-based properties and synthesise optimal strategies. We also reflect on how quantitative verification and strategy synthesis can aid or improve BDI agent system design and implementation, and propose some future work (*e.g.* multi-objective analysis).

Acknowledgements

This work is supported by the EPSRC, under PETRAS SRF grant MAGIC (EP/S035362/1), S4: Science of Sensor Systems Software (EP/N007565/1), and an Amazon Research Award on Automated Reasoning.

References

[1] Bratman, M.: Intention, plans, and practical reason. Harvard University Press (1987)
[2] Rao, A.S.: AgentSpeak (L): BDI agents speak out in a logical computable language. In: European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Springer (1996) 42–55
[3] Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative and procedural goals in intelligent agent systems. In: the 8th International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufman (2002)
[4] Hindriks, K.V., Boer, F.S.D., Hoek, W.V.d., Meyer, J.J.C.: Agent programming in 3APL. Autonomous Agents and Multi-Agent Systems **2**(4) (1999) 357–401
[5] Dastani, M.: 2APL: a practical agent programming language. Autonomous agents and multi-agent systems **16**(3) (2008) 214–248
[6] Winikoff, M.: JACK intelligent agents: an industrial strength platform. In: Multi-Agent Programming, Springer (2005) 175–193
[7] Bordini, R.H., HüJomi, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Volume 8. John Wiley & Sons (2007)
[8] Pokahr, A., Braubach, L., Jander, K.: The Jadex project: programming model. In: Multiagent Systems and Applications, Springer (2013) 21–53
[9] Wooldridge, M.: An introduction to multiagent systems. John Wiley & sons (2009)
[10] Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: A survey. ACM Computing Surveys (CSUR) **52**(5) (2019) 1–41
[11] Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Automated software engineering **19**(1) (2012) 5–63
[12] Jensen, A.B.: Machine-checked verification of cognitive agents. In: Proceedings of the 14th International Conference on Agents and Artificial Intelligence. (2022) 245–256
[13] Chen, H.: Applications of cyber-physical system: a literature review. Journal of Industrial Integration and Management **2**(03) (2017) 1750012
[14] Archibald, B., Calder, M., Sevegnani, M., Xu, M.: Probabilistic BDI agents: actions, plans, and intentions. In: Software Engineering and Formal Methods, Springer International Publishing (2021) 262–281
[15] Ghallab, M., Nau, D., Traverso, P.: Automated Planning: theory and practice. Elsevier (2004)
[16] Geffner, H., Bonet, B.: A concise introduction to models and methods for automated planning. Synthesis Lectures on Artificial Intelligence and Machine Learning **8**(1) (2013) 1–141
[17] Abdeddaı, Y., Asarin, E., Maler, O., et al.: Scheduling with timed automata. Theoretical Computer Science **354**(2) (2006) 272–300
[18] Kwiatkowska, M., Parker, D.: Automated verification and strategy synthesis for probabilistic systems. In: Automated Technology for Verification and Analysis. Springer (2013) 5–22
[19] Sardina, S., Padgham, L.: A BDI agent programming language with failure handling, declarative goals, and planning. In: Autonomous Agents and Multi-Agent Systems. Volume 23., Springer (2011) 18–70
[20] Milner, R.: The space and motion of communicating agents. Cambridge University Press (2009)

[21] Archibald, B., Calder, M., Sevegnani, M.: Probabilistic bigraphs. Formal Aspects of Computing **34**(2) (2022) 1–27

[22] Bellman, R.: A markovian decision process. Journal of mathematics and mechanics (1957) 679–684

[23] Sevegnani, M., Calder, M.: BigraphER: Rewriting and analysis engine for bigraphs. In: the 28th International Conference on Computer Aided Verification. (2016) 494–501

[24] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: the 23rd International Conference on Computer Aided Verification. Volume 6806 of LNCS., Springer (2011) 585–591

[25] Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker storm. International Journal on Software Tools for Technology Transfer (2021) 1–22

[26] Di Pierro, A., Wiklicky, H.: An operational semantics for probabilistic concurrent constraint programming. In: the 1998 International Conference on Computer Languages, IEEE (1998) 174–183

[27] Xu, M.: Bigraph models of smart manufacturing and rover example in can `https://doi.org/10.5281/zenodo.7441574` (December 2022)

[28] Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal aspects of computing **6**(5) (1994) 512–535

[29] Forejt, V., Kwiatkowska, M., Parker, D.: Pareto curves for probabilistic model checking. In: International Symposium on Automated Technology for Verification and Analysis, Springer (2012) 317–332

[30] Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)

[31] Hartmanns, A., Klauck, M.: The modest state of learning, sampling, and verifying strategies. In: International Symposium on Leveraging Applications of Formal Methods, Springer (2022) 406–432

[32] Padgham, L., Singh, D.: Situational preferences for BDI plans. In: the 2013 international conference on Autonomous agents and multi-agent systems. (2013) 1013–1020

[33] Meneguzzi, F., De Silva, L.: Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. The Knowledge Engineering Review **30**(1) (2015) 1–44

[34] De Silva, L., Meneguzzi, F.R., Logan, B.: BDI agent architectures: A survey. In: the 29th International Joint Conference on Artificial Intelligence. (2020)

[35] Bordini, R.H., Bazzan, A.L.C., Jannone, R.D.O., Basso, D.M., Vicari, R.M., Lesser, V.R.: 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) 1294–1302

[36] Xu, M., McAreavey, K., Bauters, K., Liu, W.: Intention interleaving via classical replanning. In: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence, IEEE (2019) 85–92

[37] Logan, B., Thangarajah, J., Yorke-Smith, N.: Progressing intention progression: A call for a goal-plan tree contest. In: AAMAS. (2017) 768–772

[38] Homepage: Intention progression competition. `https://www.intentionprogression.org/`

[39] Xu, M., Bauters, K., McAreavey, K., Liu, W.: A formal approach to embedding first-principles planning in BDI agent systems. In: International Conference on Scalable Uncertainty Management, Springer (2018) 333–347

A Appendix

The language used in the plan-body in Can is defined by the grammar:

$$\pm b \mid act \mid e \mid P_1; P_2 \mid P_1 \triangleright P_2 \mid P_1 \parallel P_2 \mid goal(\varphi_s, P, \varphi_f)$$

where $\pm b$ stands for belief addition and deletion, *act* a primitive agent action, and e is a sub-event (i.e. internal event). Actions *act* take the form $act = \psi \leftarrow \langle \phi^+, \phi^- \rangle$, where ψ is the pre-condition, and ϕ^+ and ϕ^- are the addition and deletion sets (resp.) of belief atoms, i.e. a belief base $\mathcal{B}$ is revised to be $(\mathcal{B} \setminus \phi^-) \cup \phi^+$ when the action executes. 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). In addition, there are composite programs $P_1; P_2$ for sequence, $P_1 \triangleright P_2$ that executes P_2 in the case that P_1 fails, 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 φ_s should be achieved through program P, failing if φ_f becomes true, and retrying as long as neither φ_s nor φ_f is true (see in [19] for details).

Fig. 7 gives the complete set of semantic rules for evolving an intention. For example, *act* handles the execution of an action, when the pre-condition ψ 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 $\triangleright_;$, $\triangleright_\top$, and $\triangleright_\bot$ 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 (within an intention). Rules G_s and G_f deal with declarative goals when either the success condition φ_s or the failure condition φ_f become true. Rule G_{init} initialises persistence by setting the program in the declarative goal to be $P \triangleright P$, *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 $G_\triangleright$ re-starts the original program if the current program has finished or got blocked (when neither φ_s nor φ_f is true).

$$\frac{act : \psi \leftarrow \langle \phi^-, \phi^+ \rangle \quad \mathcal{B} \vDash \psi}{\langle \mathcal{B}, act \rangle \rightarrow \langle (\mathcal{B} \setminus \phi^- \cup \phi^+), nil \rangle}\ act$$

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

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

$$\frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, P_1 \rhd P_2 \rangle \rightarrow \langle \mathcal{B}', P_1' \rhd P_2) \rangle}\ \rhd_; \qquad \frac{}{\langle \mathcal{B}, (nil \rhd P_2) \rangle \rightarrow \langle \mathcal{B}', nil \rangle}\ \rhd_\top$$

$$\frac{P_1 \neq nil\ \langle \mathcal{B}, P_1 \rangle \nrightarrow \ \langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}{\langle \mathcal{B}, P_1 \rhd P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}\ \rhd_\bot$$

$$\frac{\langle \mathcal{B}, P \rangle \rightarrow \langle \mathcal{B}', P' \rangle}{\langle \mathcal{B}, (nil; P) \rangle \rightarrow \langle \mathcal{B}', P' \rangle}\ ;_\top \qquad \frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, (P_1; P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1'; P_2) \rangle}\ ;$$

$$\frac{\langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1' \| P_2) \rangle}\ \|_1 \qquad \frac{\langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}{\langle \mathcal{B}, (P_1 \| P_2) \rangle \rightarrow \langle \mathcal{B}', (P_1 \| P_2') \rangle}\ \|_2$$

$$\frac{}{\langle \mathcal{B}, (nil \| nil) \rangle \rightarrow \langle \mathcal{B}, nil \rangle}\ \|_\top$$

$$\frac{\mathcal{B} \models \varphi_s}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, nil \rangle}\ G_s \qquad \frac{\mathcal{B} \models \varphi_f}{\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, ?false \rangle}\ G_f$$

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

$$\frac{\mathcal{B} \not\models \varphi_s \quad \mathcal{B} \not\models \varphi_f \quad \langle \mathcal{B}, P_1 \rangle \rightarrow \langle \mathcal{B}', P_1' \rangle}{\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f) \rangle \rightarrow \langle \mathcal{B}', goal(\varphi_s, P_1' \rhd P_2, \varphi_f) \rangle}\ G_;$$

$$\frac{\mathcal{B} \not\models \varphi_s \quad \mathcal{B} \not\models \varphi_f \quad \langle \mathcal{B}, P_1 \rangle \nrightarrow}{\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f) \rangle \rightarrow \langle \mathcal{B}, goal(\varphi_s, P_2 \rhd P_2, \varphi_f) \rangle}\ G_\rhd$$

Fig. 7. Complete intention-level CAN semantics.