Modelling and Verifying BDI Agents with Bigraphs

Blair Archibald[a], Muffy Calder[a], Michele Sevegnani[a], Mengwei Xu[a,*]

[a]*School of Computing Science, University of Glasgow, UK*

Abstract

The Belief-Desire-Intention (BDI) architecture is a popular framework for rational agents; existing verification approaches either directly encode simplified (e.g. lacking features like failure recovery) BDI languages into existing verification frameworks (e.g. Promela), or reason about specific BDI language *implementations*. We take an alternative approach and employ Milner's bigraphs as a modelling framework for a fully featured BDI language, the Conceptual Agent Notation (CAN)—a superset of AgentSpeak featuring declarative goals, concurrency, and failure recovery. We provide an encoding of the syntax and semantics of CAN agents, and give a rigorous proof that the encoding is faithful. Verification is based on the use of mainstream software tools including BigraphER, and a small case study verifying several properties of Unmanned Aerial Vehicles (UAVs) illustrates the framework in action. The *executable* framework is a foundational step that will enable more advanced reasoning such as plan preference, intention priorities and trade-offs, and interactions with an environment under uncertainty.

Keywords: BDI Agents, Modelling, Verification, Bigraphs

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. BDIs have been very successful in many areas such as business [7], healthcare [8], and engineering [9].

The deployment of autonomous systems in real-world applications raises concerns of trustworthiness and safety, for example in scenarios such as autonomous

*Corresponding Author

Email addresses: `blair.archibald@glasgow.ac.uk` (Blair Archibald), `muffy.calder@glasgow.ac.uk` (Muffy Calder), `michele.sevegnani@glasgow.ac.uk` (Michele Sevegnani), `mengwei.xu@glasgow.ac.uk` (Mengwei Xu)

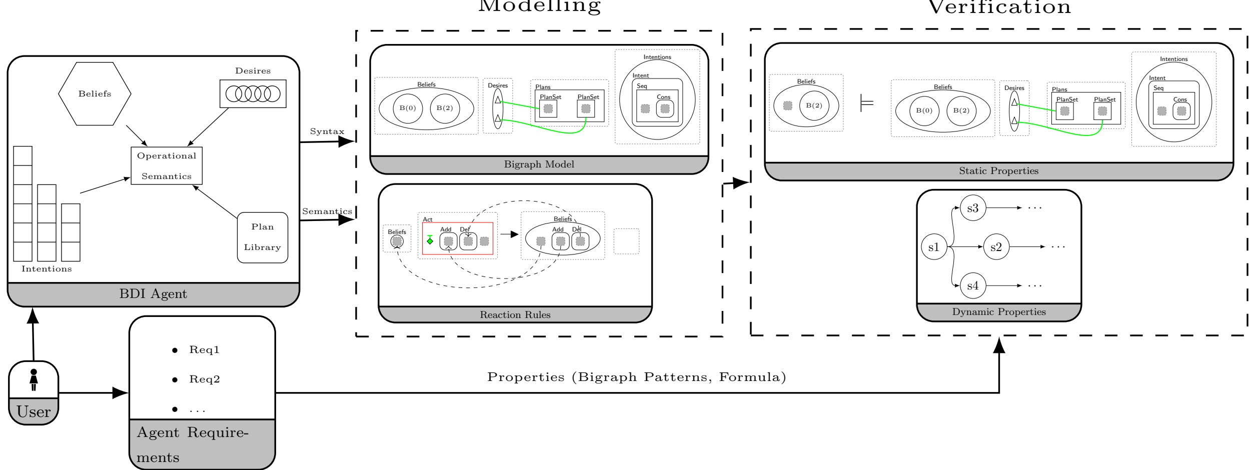


Figure 1: Modelling and verification framework for BDI agents.

control in space [10] and human-robot interaction in healthcare [11]. There is a growing demand for verification techniques to aid analysis of behaviours in increasingly complex and critical domains, and there has been a proliferation of techniques and languages supporting BDI agent verification. Most of these approaches either i) encode simplified, e.g. lacking features like failure recovery, BDI languages directly into verification frameworks/model checkers, for example, translating a simplified AgentSpeak language [12] into Promela/Spin [13, 14] (no translation proof given) or ii) focus on verifying a specific *implementation* of a BDI programming language, for example the Agent Infrastructure Layer (AIL) [15] implements a BDI language as a set of Java classes that can be verified with the Java PathFinder [16] program model checker. While verifying an implementation tells you how the system *will* operate, it might not correspond to how it *should* operate with respect to the semantics of the given BDI programming language.

We present an approach for reasoning about the semantics of a fully-fledged BDI programming language— with the implementation of a faithful semantics encoding and incorporation of some advanced BDI features—through a mathematical model of agent execution, i.e. verified executable semantics.

Our approach models and verifies BDI agents, specified in the Can language, by encoding them as an instance of Milner's Bigraphical Reactive Systems (BRS) [17]. Can 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 AgentSpeak, Can includes advanced BDI agent behaviours such as reasoning with *declarative goals*, *concurrency*, and *failure recovery*. Importantly, although we focus on Can, the language features are similar to those of other mainstream BDI languages and the same modelling techniques would apply to other BDI programming languages.

Bigraphs provide a meta-modelling framework, developed as a unifying theory for calculi, e.g. π-calculus [18], with extensions for priority and conditional rewriting [19, 20]. As a graph-based rewriting formalism, over rules called reaction rules, bigraphs not only provide an intuitive diagrammatic representation, which is ideal for visualising the execution process of CAN, but also offer compositional reasoning via explicit abstractions (sites/regions/names), customised rewriting rules, and multiple ways to relate entities (placement and linking). While rewriting based approaches have previously been used for agent systems [21], they are based on term, rather than graph, rewriting.

For analysis, BigraphER [19] is a freely available tool including rewriting, verification based on bigraph patterns, and transition system export to model checking tools [22].

Our bigraph encoding of the CAN language includes: i) a structural encoding that maps the syntax of CAN (e.g. beliefs, plans, and intentions) into equivalent bigraphs, and ii) an encoding of the operational semantics of CAN as a set of reaction rules. We provide a correctness proof that the translation of CAN semantics into reaction rules is *faithful*.

The framework is depicted in Fig. 1. On the left we have the BDI agents and agent requirements – logical formulas. In the middle, Modelling, the agents are translated into bigraphs that capture the structural elements and reaction rules that capture their dynamics. Once we have encoded the agent into bigraphs, we use the model to perform verification (on the right) of user-specified *agent requirements*. Verification takes two forms: checking static properties of a state (the current bigraph representing an agent at some point in its execution) through bigraph patterns, and checking dynamic properties, expressed as temporal logic properties, against the transition system generated by BigraphER. Finally, the user can employ BigraphER simply to "run" their agent model with different initial settings.

We illustrate the framework with a small case study based on Unmanned Aerial Vehicles (UAVs).

We make the following contributions:

- an encoding of the CAN language and operational semantics in bigraphs, using regions to represent the perspectives of *Belief*, *Desire*, *Intention*, and *Plan*,
- proof that the encoding is faithful by showing each CAN semantic rule is encoded by a (finite) sequence of reaction rules,
- an illustration of our framework in a UAV case study,
- a reflection on CAN based on insights gained from the encoding and also reflections on the experience, both theoretical and practical, of bigraphs as an approach to reason about agent programming languages—and programming languages more generally,
- an overview of how we will build upon this foundation in the future to

reason about plan selection, intention tradeoffs and priorities, and interactions with an uncertain environment.

The paper is organised as follows: in Section 2, we recall preliminaries of BDI agents in the CAN language and bigraphs; in Section 3, we provide the structural encoding that maps the syntax of CAN into equivalent bigraphs; in Section 4, we present a comprehensive review of the core semantics of CAN (excluding concurrency and declarative goals) and, in particular, how the operation of CAN semantics can be viewed as AND/OR trees. In Section 5 we encode the semantics given in Section 4 and in Section 6, we present the semantics for concurrency and declarative goals and provide their bigraph encodings. In Section 7, we illustrate our framework with examples and in Section 8, we reflect on aspects of CAN. In Section 9 we discuss related work; in Section 10 we lay out the our plans of for future extensions to this work; we conclude in Section 11.

2. Preliminaries

We give an overview of BDI agents, described in the Conceptual Agent Notation (CAN) language, as well as Bigraphs and Bigraphical Reactive Systems (BRS).

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.

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. In the following, we introduce the syntax of CAN, the semantics is given in Section 4.

A CAN agent consists 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. Without loss of generality, we specify our belief base following the logical language in AgentSpeak [2] that takes the form $\varphi ::= b \mid \neg b \mid (\varphi_1 \wedge \varphi_2) \mid \mathit{true} \mid \mathit{false}$ (where b denotes a ground belief atom). More complex logics are possible but are outwith the scope of this paper, i.e. we show how to encode general BDI agents in bigraphs, not how to encode specific logics. All that we assume for any chosen logical language is that it has belief operators to check whether a belief formula φ follows from the belief base (i.e. $\mathcal{B} \models \varphi$), 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* Π 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, φ the context condition, and P the plan-body. The

triggering event e specifies why the plan is triggered, the context condition φ determines *when* the plan-body P is able to handle the event. We denote the triggering event of a plan Pl $trigger(Pl)$ and we call $E = \{trigger(Pl) \mid Pl \in \Pi\}$ the *event set* that the agent knows how to respond to (i.e. it has plans for response – though it might be the case none are applicable). 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 φ 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 $act = \varphi \leftarrow \langle \phi^+, \phi^- \rangle$, where φ 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 with addition and deletion sets ϕ^+ and ϕ^- 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 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 [23] 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 Γ (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).

2.1.1. Running Example – Conference Travel Agent

For illustration, we give a classic example—arranging a conference trip—as shown in Table 1.

A BDI agent desires to arrange a conference trip, denoted by an external event e_1. We assume there are only two ways to travel to the conference. The first way is to travel by car, given by the plan $Pl_1 = e_1 : b_1 \wedge b_2 \leftarrow act_1; act_2$. The plan Pl_1 expresses that if the agent believes it owns a car (i.e. b_1) and the venue is in the driving distance (i.e. b_2), it can start the car and drive all the way to the venue. To specify the actions, we have $act_1 = b_3 \leftarrow \langle \{b_4\}, \emptyset \rangle$ and $act_2 = b_4 \leftarrow \langle \{b_5\}, \emptyset \rangle$. For example, the action act_1 expresses that if the car is functional (i.e. b_3) and after executing act_1, the belief of the engine being on (i.e. b_4) will be added while deleting nothing from the belief base.

Table 1: A BDI Agent for Conference Travelling.

Belief base	**External events**	**Plan library**	**Actions**
b_1	e_1	$Pl_1 = e_1 : b_1 \wedge b_2 \leftarrow act_1; act_2$	$act_1 = b_3 \leftarrow \langle\{b_4\}, \emptyset\rangle$
b_2		$Pl_2 = e_1 : b_6 \wedge b_7 \leftarrow act_3; e_2; act_4$	$act_2 = b_4 \leftarrow \langle\{b_5\}, \emptyset\rangle$
b_3		$Pl_3 = e_2 : b_8 \leftarrow act_5; act_6$	$act_3 = true \leftarrow \langle\{b_8\}, \emptyset\rangle$
b_4			$act_4 = b_9 \leftarrow \langle\{b_5\}, \emptyset\rangle$
			$act_6 = b_{10} \leftarrow \langle\{b_9\}, \{b_8, b_{10}\}\rangle$

where e_1 stands for `conference_travelling`, e_2 for `get_onboard`, act_1 for `start_car`, act_2 for `driving`, act_3 for `book_flight`, act_4 for `go_to_venue`, act_5 for `go_to_airport`, act_6 for `flying`, b_1 for `own_car`, b_2 for `driving_distance`, b_3 for `car_functional`, b_4 for `engine_on`, b_5 for `at_venue`, b_6 for `budget_allowed`, b_7 for `flight_available`, b_8 for `flight_booked`, b_9 for `flight_landed`, and b_{10} for `at_airport`.

The second way is to travel by air, given by the plan $Pl_2 = e_1 : b_6 \wedge b_7 \leftarrow act_3; e_2; act_4$. This plan expresses that if the budget allows (i.e. b_6) and there is a flight (i.e. b_7), the agent can book the ticket first, then post internally a sub-event to actually travelling by plane, and go to the venue after landing. For actions, we have $act_3 = true \leftarrow \langle\{b_8\}, \emptyset\rangle$ and $act_4 = b_9 \leftarrow \langle\{b_5\}, \emptyset\rangle$. To address the sub-event e_2, we have plan $Pl_3 = e_2 : b_8 \leftarrow act_5; act_6$. Pl_3 expresses that if the agent believes the flight has been booked, it can go to the airport and fly by plane. Also, we have $act_5 = b_8 \leftarrow \langle\{b_{10}\}, \emptyset\rangle$ and $act_6 = b_{10} \leftarrow \langle\{b_9\}, \{b_8, b_{10}\}\rangle$. In particular, action act_6 indicates that if at airport (i.e. b_{10} for `at_airport`), after the flight it will add the belief atom b_9 for `flight_landed`, and delete both belief atoms b_8 for `flight_booked` and b_{10} for `at_airport`.

We define the initial belief base to be $\mathcal{B} = \{b_1, b_2, b_6, b_7\}$. This expresses the agent believes that it owns a car (b_1), the venue is in the driving distance (b_2), the budget is sufficient for flight (b_6), and there is a flight available (b_7).

2.2. Bigraphs

Bigraphs are a universal modelling language, introduced by Milner [24], for both modelling ubiquitous systems and as a unifying theory for many existing calculi for concurrency and mobility. A bigraph consists of a pair of relations over the same set of *entities*: a directed forest representing topological space in terms of containment, and a hyper-graph expressing the interactions and (non-spatial) relationships among entities. Each entity is assigned a *type*, which determines its *arity* (i.e. number of links), and whether it is *atomic* (i.e. it cannot contain other entities). For the purpose of presenting our approach, we provide only an informal overview of bigraphs. The full theory is detailed elsewhere [24].

Bigraphs can be described in algebraic terms or with an *equivalent* diagrammatical representation as shown in Table 2. An example bigraph, representing a simple phone connection and cloning scenario, is in Fig. 2a. In general, bigraphs permit any kind of shape (sometimes coloured) for typed entities, e.g. we use a lock symbol for entity Locked and a diamond for entity Data. We allow entities to be *parameterised*, i.e. $\mathsf{K}(n)$ for $n \in \mathbb{N}$, allowing them to represent *families* of entities, e.g. $\mathsf{K}(0), \mathsf{K}(1), \ldots$. Entities can be connected through green links.

Table 2: Bigraph components and operations.

Component/Operation	Algebraic Form	Diagrammatic Form
Entity of arity 1	K_a	
Name closure	$/a\ \mathsf{K}_a$	
Site	id	
Region	1	
Nesting	$\mathsf{Act.B.id}$	
Parallel product	$\mathsf{C}_x.\mathsf{id} \parallel \mathsf{D}_x.\mathsf{id}$	
Merge product	$\mathsf{C}_x.\mathsf{id} \mid \mathsf{D}_x.\mathsf{id}$	

Names[1] allow links (or potential links) to bigraphs in an external environment or context, and are written above the bigraph. Unconnected links are *closed* and drawn as a closed-off link. Grey rectangles are called *sites* that indicate parts of the model that have been abstracted away. In other words, an entity containing a site can contain zero or more entities of any kind. Finally, a dashed rectangle denotes a *region* of adjacent parts of the system.

Topological placement of entities—given by the place-graph of Fig. 2b—is described using: *nesting* that defines the containment relation on entities; *merge product* that places two entities side-by-side at the *same* hierarchical level (e.g. the two Room entities as they share a common parent); and *parallel product* that places entities in separate regions, allowing them to be at different levels of the hierarchy (e.g. any two Phone entities). Importantly, parallel product would *not* match where one is below the other in the place graph, i.e. we are looking for two disjoint sub-trees in the place graph. In both merge and parallel product, bigraphs are linked on common names allowing a link graph (shown in Fig. 2c) to be constructed, e.g. two Phone entities in different regions can still connect. An overview of the bigraph components and operations are given in Table 2.

For the example of Fig. 2, we can equivalently write it using the algebraic

[1] Specifically outer-names.

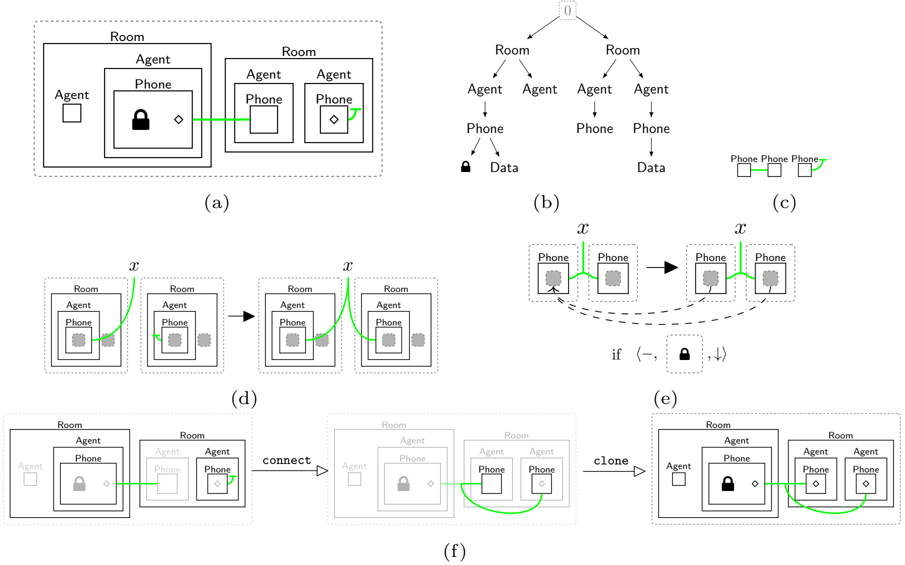


Figure 2: Full bigraph modelling example: Agents and Calls. (a) Initial bigraph; (b) Place graph; (c) Link graph; (d) Reaction rule `connect`; (e) Reaction rule `clone`; (f) Possible execution trace (matches shown as bold).

notion as:

$$/l\ (\mathsf{Room}.(\mathsf{Agent}.\mathsf{Phone}_l.(\mathsf{Locked} \mid \mathsf{Data}) \mid \mathsf{Agent}.1) \mid \\ \mathsf{Room}.(\mathsf{Agent}.\mathsf{Phone}_l.1 \mid \mathsf{Agent}.(/c\ \mathsf{Phone}_c.\mathsf{Data}))$$

2.3. Bigraphical Reactive Systems

A bigraph represents a system at a single point in time. To allow models to evolve over time we can specify a Bigraphical Reactive System (BRS) that acts as a rewriting system. A BRS consists of a set of *reaction rules* of the form $L \longrightarrow R$, where L and R are bigraphs. Intuitively, a bigraph B evolves to B' by matching and rewriting an occurrence of L in B with R. Such a reaction is indicated with $B \longrightarrow B'$. We use $\longrightarrow^+$ to denote *one* or more applications of a rule, and $\longrightarrow^*$ to denote *zero* or more rule applications. We also write $\xrightarrow{\texttt{rule}}$ to identify the reaction rule being applied to generate the transition. If no name is specified we assume any rule applied. Reaction rules can be parameterised when they are defined over entities with parameterised types, i.e. a rule $\texttt{r}(k)$ for all values of k. The *transition system* of a BRS is a (possibly infinite) graph whose vertices are bigraphs representing the reachable states and whose edges represent reactions over bigraphs.

BRSs are closely related to term rewriting [25], with bigraphs as terms and model semantics determined by a set of *user-specified* rewrite rules. The only built-in BRS semantic is matching a (sub-)bigraph and rewriting with a new

bigraph. This is in contrast to other modelling formalisms, e.g. π-calculus, that use a fixed set of semantic rules and the user specifies a model that, when executed on those rules, performs the expected operations. Term rewriting such as Maude language [21] has found use in modelling agents previously, and bigraphs offer similar advantages, while benefiting from e.g. the intuitive diagrammatic notion, and support for multiple modelling dimensions (place and link).

An example reaction rule, `connect`, is in Fig. 2d, which models the case where a *disconnected* Phone wants to connect to a call. The use of a name means the first Phone may be *either* already on a call (creating a conference call), or itself disconnected. By explicitly matching on Room entities we force the agents in the reaction to be in two different rooms, although they might connect (through x) to another Phone in the same room. An example of applying `connect` to the example bigraph of Fig. 2a is the first transition of Fig. 2f.

We also use conditional bigraphs [20] that allow *application conditions* to specify contextual requirements within the rewrite system. For example, we can exclude certain bigraphs appearing within sites of the left-hand-side of a rule. We write conditions in the form: if $\langle -, \bullet, \downarrow \rangle$ where the $-$ indicates a negative condition i.e. that the bigraph of the condition should *not* appear/be matched, the black circle represents an arbitrary bigraph we want to ensure does not appear, and $\downarrow$ indicates we specifically do not want the condition to appear in (any of) the sites[2]. Importantly the bigraph in the condition cannot appear *anywhere* in the site, including nested below other entities. When more than one condition is specified for a reaction (separated by commas) they must *all* hold for the rule to apply.

An example reaction rule using conditional bigraphs is in Fig. 2e, which shows how a Phone may be cloned so long as *neither* Phone contains a lock, i.e. a Locked entity is nowhere in the two sites. To allow copying (and deletion), reaction rules can be augmented with *instantiation maps* that determine a mapping between sites on the left and right-hand side of a reaction rule. Instantiation maps are denoted graphically as dashed arrows mapping sites in the right-hand side R to sites in left-hand side L. The instantiation map is omitted from a rule definition when it is an identity. For example, in Fig. 2e, the instantiation map forces the two sites in the right hand side to be copies of the first site on the left. An example of applying `clone` is the second transition in Fig. 2f. Here, due to the commutative nature of parallel product, we match the Phone containing data as the first Phone in the rule and the empty Phone as the second. This rule would not apply to the locked phone due to the condition.

Furthermore, rule *priorities* can be introduced by defining a partial ordering on the reaction rules of a BRS, as implemented in [26]. A reaction rule of lower priority can be applied only if no rule of higher priority is applicable. We write $\mathtt{r}_1 < \mathtt{r}_2$ when $\mathtt{r}_2$ has higher priority than $\mathtt{r}_1$. This notation extends to sets in the natural manner, e.g. $\{\mathtt{r}_1, \mathtt{r}_3\} < \{\mathtt{r}_2, \mathtt{r}_4\}$, where rules in the same set have

[2]Conditional bigraphs also allows positive, and contextual conditions, however we do not use these here.

the same priority.

A common approach for verifying a BRS is through (bounded) model checking on its transition system, e.g. in Fig. 2f. To allow labelling of states, which are themselves bigraphs, we define predicates as *bigraph patterns*. Informally, a pattern can be seen as a left-hand-side of a reaction rule, i.e. the input to the matching problem. A single state may have multiple labels if multiple patterns occur in it. Patterns can also be combined with standard Boolean operators to form logical formulae.

3. Encoding BDI Agents in Bigraphs

We define the structural encoding that maps the syntax (e.g. plans and actions) of a CAN BDI agent into equivalent bigraphs.

Recall a BDI agent is specified by a belief base $\mathcal{B}$ consisting of a set of belief atoms, e.g. $\mathcal{B} = \{b_1, ..., b_n\}$, a set of events (i.e. desires) the agent responds to, and a plan library Π containing plans in form of $Pl = e : \varphi \leftarrow P$. As the agent executes, plan-bodies selected for addressing desires become the intentions of the agent.

We take a *multi-perspective* approach (as introduced in [22]) in which perspectives are represented by separate and parallel regions. Mirroring the core components of a BDI agent, we employ four perspectives: *Belief* that handles knowledge storage and updates; *Desire* that manages the external events; *Intention* that captures the current execution states of plan-bodies; and *Plan* that holds instructions for the agent on how to bring about its desires (i.e. how to respond to specific events). This approach allows us to separate design concerns, to be explicit how and when concerns interact, and to visualise them naturally, as shown in Fig. 1. It also facilitates model extension, for example we could in future add perspectives for the external (uncertain) environment, or we could *replace* the Beliefs perspective with one that allows more complex logic formulas.

The entities in the bigraph model for the syntax of a BDI agent are given in Table 3, grouped by the four perspectives. For each entity we give the algebraic form as well as structural information in the form of valid parents and linked entities. The only atomic entities, i.e. that cannot nest other entities, are belief atoms $\mathsf{B}(n)$, logical constant e.g. false, and events E_e. Detailed information on the role for each of these entities is given as we introduce the encoding.

We define an encoding $[\![\cdot]\!] : BDI \rightarrow \mathbf{Bg}(\mathcal{K})$ that maps the syntax of a BDI agent – including beliefs, desires, intentions, and plans – to an equivalent bigraph, where $\mathcal{K}$ denotes the set of all entity types in Table 3. No information is lost through $[\![\cdot]\!]$ and it is possible to define the inverse encoding $[\![\cdot]\!]^{-1}$ establishing an (structural) equivalence, that is, for any agent A we have $A = [\![[\![A]\!]]\!]^{-1}$ as required. Although the inverse is easy to define, some cases are context dependent, e.g. Eq. (12) and Eq. (13) related to belief atoms in Fig. 3 have the same bigraph representation but always appear in distinct contexts (pre-

Table 3: Bigraph entities for BDI syntax encoding.

Description	Entity	Parent(s)	LinksTo	Diagrammatic Form
Belief Base	Beliefs			Beliefs
Belief Atoms	B(n)	{Beliefs, Pre, Add, Del}		B(n)
Logical False	false	{Beliefs, Pre}		false
Desire Set	Desires			Desires
Event	E$_e$	{Desires, PB, Conc}	PlanSet$_e$	
Intention base	Intentions			Intentions
Intention	Intent	Intentions		Intent
Plan library	Plans			Plans
Relevant Plans	PlanSet$_e$	{Plans, Intent, Seq, Cons, L, R}	E$_e$	PlanSet
Plan	Plan	PlanSet$_e$		Plan
Plan Body	PB	Plan		PB
Action	Act	{PB, Seq, Cons, L, R}		Act
Precondition	Pre	{Act, Plan}		Pre
Belief Addition	Add	Act		Add
Belief Deletion	Del	Act		Del
Sequence ;	Seq	{PB, Try}		Seq
Plan Choice ▷	Try	{Intent, Seq, Goal, L, R}		Try
Next Pointer	Cons	{Seq, Try}		Cons
Concurrency ‖	Conc	{Seq, Try, PB}		Conc
Concurrency Markers	{L, R}	Conc		L
Declarative Goal	Goal	{Seq, Try, L, R, PB}		Goal
Success Condition	SC	Goal		SC
Failure Condition	FC	Goal		FC

$$\llbracket b_n \rrbracket = \mathsf{B}(n) \tag{1}$$
$$\llbracket false \rrbracket = \mathsf{false} \tag{2}$$
$$\llbracket true \rrbracket = 1 \tag{3}$$
$$\llbracket \mathcal{B} = \{b_1 \dots b_n\} \rrbracket = \mathsf{Beliefs}.(\llbracket b_1 \rrbracket \mid \dots \mid \llbracket b_n \rrbracket) \tag{4}$$
$$\llbracket E^e = \{e_1 \dots e_n\} \rrbracket = \mathsf{Desires}.(\llbracket e_1 \rrbracket \mid \dots \mid \llbracket e_n \rrbracket) \tag{5}$$
$$\llbracket e \rrbracket = \mathsf{E}_e \tag{6}$$
$$\llbracket \Gamma = \{P_1, ..., P_n\} \rrbracket = \mathsf{Intentions}.(\mathsf{Intent}.\llbracket P_1 \rrbracket \mid \cdots \mid \mathsf{Intent}.\llbracket P_n \rrbracket) \tag{7}$$
$$\llbracket \Pi = \{Pl_1 \dots Pl_n\} \rrbracket = \mathsf{Plans}.\prod_{e \in E} \mathsf{PlanSet}_e.(\llbracket Pl_j \rrbracket \mid \dots \mid \llbracket Pl_k \rrbracket)_{\text{where } trigger(Pl_j) = trigger(Pl_k) = e} \tag{8}$$
$$\llbracket \langle N_1, \dots, N_n \rangle \rrbracket = \llbracket N_1 \rrbracket \parallel \dots \parallel \llbracket N_n \rrbracket \text{ where } N_i \in \{E^e, P, \mathcal{B}, \Gamma, \Pi\} \tag{9}$$

(a) Beliefs, desire, intention, and plan library encoding.

$$\llbracket nil \rrbracket = 1 \tag{10}$$
$$\llbracket act = \varphi \leftarrow \langle \phi^+, \phi^- \rangle \rrbracket = \mathsf{Act}.(\mathsf{Pre}.\llbracket \varphi \rrbracket \mid \mathsf{Add}.\llbracket \phi^+ \rrbracket \mid \mathsf{Del}.\llbracket \phi^- \rrbracket) \tag{11}$$
$$\llbracket \varphi = b_1 \wedge \cdots \wedge b_n \rrbracket = \llbracket b_1 \rrbracket \mid \dots \mid \llbracket b_n \rrbracket \tag{12}$$
$$\llbracket \phi^\pm = \{b_1 \dots b_n\} \rrbracket = \llbracket b_1 \rrbracket \mid \dots \mid \llbracket b_n \rrbracket \tag{13}$$
$$\llbracket P_1 ; P_2 \rrbracket = \mathsf{Seq}.(\llbracket P_1 \rrbracket \mid \mathsf{Cons}.\llbracket P_2 \rrbracket) \tag{14}$$
$$\llbracket P_1 \parallel P_2 \rrbracket = \mathsf{Conc}.(\mathsf{L}.\llbracket P_1 \rrbracket \mid \mathsf{R}.\llbracket P_2 \rrbracket) \tag{15}$$
$$\llbracket goal(\varphi_s, P, \varphi_f) \rrbracket = \mathsf{Goal}.(\mathsf{SC}.\llbracket \varphi_s \rrbracket \mid \llbracket P \rrbracket \mid \mathsf{FC}.\llbracket \varphi_f \rrbracket) \tag{16}$$
$$\llbracket P_1 \rhd P_2 \rrbracket = \mathsf{Try}.(\llbracket P_1 \rrbracket \mid \mathsf{Cons}.\llbracket P_2 \rrbracket) \tag{17}$$
$$\llbracket e : (|\varphi_1 : P_1, \dots, \varphi_2 : P_2|) \rrbracket = \mathsf{PlanSet}_e.(\llbracket \varphi_1 : P_1 \rrbracket \mid \dots \mid \llbracket \varphi_2 : P_2 \rrbracket) \tag{18}$$
$$\llbracket \varphi : P \rrbracket = \mathsf{Plan}.(\mathsf{Pre}.\llbracket \varphi \rrbracket \mid \mathsf{PB}.\llbracket P \rrbracket) \tag{19}$$
$$\llbracket Pl = e : \varphi \leftarrow P \rrbracket = \llbracket \varphi : P \rrbracket \tag{20}$$

(b) Plan and plan-body encoding.

Figure 3: Encoding $\llbracket \cdot \rrbracket$ from BDI agents to bigraphs.

conditions and action outcomes respectively)[3]. For brevity we omit the details of the inverse encoding.

The encoding is defined inductively as shown in Fig. 3. To aid explanation, we give the encoding in two parts. In Fig. 3a, the encoding of agent belief, desire and intention structures are given. In the second part, the encoding of plans, in particular the plan-bodies, of an agent are provided in Fig. 3b. The parts are not distinct e.g. the plans within the plan library are encoded using the encoding of plan-bodies. We use $\prod \mathsf{M} \stackrel{\text{def}}{=} \mathsf{M} \mid \dots \mid \mathsf{M}$ to denote iterated merge product. In the next few sections, we explain the (numbered) equations in Fig. 3.

[3]The inverse of true is a special case as we may map either to the truth term or an empty context set. However both options give rise to behaviourally equivalent agents.

3.1. Encoding of Beliefs, Desires, and Intentions

Equation 9 is a general rule describing how tuples map into parallel regions. We use this to ensures the top-level components of an agent $\langle \mathcal{B}, E^e, \Gamma, \Pi \rangle$—beliefs, desires, intentions, and plan library—are mapped to separate perspectives (regions) in the bigraph.

We assume all belief formulas φ are expressed in propositional logic. Recall that the convention in AgentSpeak for the belief base is $\varphi ::= b \mid \neg b \mid (\varphi_1 \wedge \varphi_2) \mid \mathit{true} \mid \mathit{false}$. For convenience, the *parameterised entities* $\mathsf{B}(n)$ (Eq. (1)) are used for both positive and negative atoms: b or $\neg b$ (e.g. $[\![b]\!] = [\![b_0]\!] = \mathsf{B}(0), [\![\neg b]\!] = [\![b_1]\!] = \mathsf{B}(1)$). Using this, all formulas can be constructed in pure conjunctive form, i.e. $\varphi = b_1 \wedge \cdots \wedge b_n$. We allow logical constants for $true$ and $false$ representing formulas that are always/never entailed, e.g. an action with pre-condition $false$ never executes. In the bigraph model, we only assign an entity false to represent logical $false$ (Eq. (2)), while the logical constant $true$ is mapped to the empty bigraph (Eq. (3)) as we assume an empty formula is always true, e.g. there may be no pre-condition for some action.

Encoding the belief base $\mathcal{B}$ (and any set concept in general) from a BDI agent to bigraphs leverages the bag-like nature of nesting (Eq. (4)). For empty sets, we have $[\![\emptyset]\!] = 1$, i.e. the bigraph with one empty region.

To encode desires, an entity of E_e is created for each possible event (that an agent *desires* to respond to) as seen in Eq. (5) and Eq. (6). Importantly, E_e exports a *name* e that allows us to identify specific events using links. Recall that a set of relevant plans is the set of plans which have the same triggering event. We use this when encoding the plan library Π (Eq. (8)) by having it contain sets of relevant plans $\mathsf{PlanSet}_e$ with e connecting the event e with the set of plans that respond to it. This differs from typical BDI agents where the plan library is a flat set of plans. This use of indexing by event name through relevant plans decreases the likelihood of some potential human errors, e.g. misspelling of event names and also simplifies agent reasoning by avoiding repetitive searching for relevant plans.

Finally, for intentions, we utilise the same set-like structure as beliefs, this time encoding individual (partially executed) plan-bodies as required (Eq. (7)), which will be discussed in the next section.

3.2. Encoding Plans and Plan-Bodies

Plans and plan-bodies are specified with the language given in Fig. 4, which includes two forms of plan-bodies: $\langle$UserP$\rangle$ that the user writes, and the more comprehensive $\langle$P$\rangle$ that can occur during any execution.

A plan e : Pre $\leftarrow$ $\langle$UserP$\rangle$ consists of a triggering event e, the context (pre-condition) $\langle$Pre$\rangle$, and a user-defined plan-body specified by $\langle$UserP$\rangle$. The user-defined plan-body $\langle$UserP$\rangle$ may be the basic building block $\langle$BasicP$\rangle$ including handling an internal event e, or executing an action $\langle$Act$\rangle$. Actions also have the pre-condition $\langle$Pre$\rangle$, which indicates when an action is valid for execution given in the current belief state. After executing an action, ϕ^+ and ϕ^- are sets of beliefs to be added and removed from the belief state, respectively.

$$
\begin{array}{rcl}
\langle \text{Plan} \rangle & ::= & \mathtt{e} : \langle \text{Pre} \rangle \leftarrow \langle \text{UserP} \rangle \\
\langle \text{UserP} \rangle & ::= & \langle \text{BasicP} \rangle \mid \langle \text{UserP} \rangle ; \langle \text{UserP} \rangle \mid \langle \text{UserP} \rangle \| \langle \text{UserP} \rangle \mid \\
& & goal(\varphi_s, \mathtt{e}, \varphi_f) \\
\langle \text{P} \rangle & ::= & \mathtt{nil} \mid \langle \text{BasicP} \rangle \mid \langle \text{P} \rangle ; \langle \text{P} \rangle \mid \langle \text{P} \rangle \| \langle \text{P} \rangle \mid \\
& & goal(\varphi_s, \langle \text{P} \rangle, \varphi_f) \mid \langle \text{P} \rangle \rhd \langle \text{P} \rangle \mid \\
& & \mathtt{e} : (|\varphi_1 : \langle \text{UserP} \rangle, \ldots, \varphi_n : \langle \text{UserP} \rangle|) \\
\langle \text{BasicP} \rangle & ::= & \mathtt{e} \mid \langle \text{Act} \rangle \mid +b \mid -b \mid ?\varphi \\
\langle \text{Act} \rangle & ::= & \langle \text{Pre} \rangle \leftarrow \langle \phi^+, \phi^- \rangle \\
\langle \text{Pre} \rangle & ::= & \varphi \mid false \mid truth
\end{array}
$$

Figure 4: Grammar for plans and plan-bodies.

The user-defined plan-body ⟨UserP⟩ can also be combined in the three ways: ⟨UserP⟩; ⟨UserP⟩ executing those two ⟨UserP⟩ in sequence, ⟨UserP⟩ ‖ ⟨UserP⟩ pursing those two ⟨UserP⟩ concurrently, and $goal(\varphi_s, e, \varphi_f)$ achieving the state φ_s through addressing an internal event e, failing when φ_f holds, and retrying as long as neither φ_s nor φ_f is believed to be true. Internally (i.e. during execution) programs may have three additional forms: *nil* is the empty program that is always successful, ⟨P⟩ ▷ ⟨P⟩ represents *trying* the first ⟨P⟩ while keeping the second ⟨P⟩ as a backup in case the first ⟨P⟩ fails, and $\mathtt{e} : (|\varphi_1 : \langle \text{UserP} \rangle \ldots \varphi_n : \langle \text{UserP} \rangle|)$ is a set of backup plans which are all triggered by the event e.

The bigraph encoding of plans and plan-bodies (Fig. 3b) mirrors the grammar given in Fig. 4 by specifying a mapping for each syntactic form. Each individual plan is represented as the pairing of some pre-condition (as encoded belief atoms), nested in the entity Pre, and an encoded plan-body, nested in entity PB (Eq. (19) and Eq. (20) in Fig. 3b).

Bigraph entities of ⟨UserP⟩ are built by introducing additional controls for each form, e.g. Seq. As the merge product operator of bigraphs is commutative, i.e. $A \mid B \equiv B \mid A$, we need to add additional entities to force an ordering on the children. For example, the sequencing $P_1; P_2$ (Eq. (14) in Fig. 3b) utilises an entity Cons that identifies P_2 as the next to execute after the successful execution of its predecessor P_1. Likewise, the form $P_1 \rhd P_2$ (Eq. (17)), that *tries* P_1 with P_2 as a backup, uses Cons to distinguish between P_1 and P_2. For concurrency (Eq. (15)) we require two additional controls L and R to identify the left and right of the concurrency structure ‖. Finally, for the form of declarative goals $goal(\varphi_s, P, \varphi_f)$ (Eq. (16)), we map it to an entity Goal that nests a success condition SC, failure condition FC, and the current form of the remaining program.

Finally, actions are encoded (Eq. (11)) in a similar way. In particular, raw entailment and belief state update forms, i.e. $?\varphi, +b$, and $-b$, may be seen as special cases of actions that do not update the external environment. We estab-

Table 4: Example encoding of a conference travel agent in Table 1.

Agent	$[\![$**Agent**$]\!]$
$\mathcal{B} = \{b_1, b_2, b_6, b_7\}$	$\mathsf{Beliefs}_{.}(\mathsf{B}(1) \mid \mathsf{B}(2) \mid \mathsf{B}(6) \mid \mathsf{B}(7))$
$\Pi = \{Pl_1, Pl_2, Pl_3\}$	$\mathsf{Plans}_{.}(\mathsf{PlanSet}_{e_1}.([\![Pl_1]\!] \mid [\![Pl_2]\!]) \mid \mathsf{PlanSet}_{e_2}.[\![Pl_3]\!])$
$Pl_1 = e_1 : \varphi_1 \leftarrow act_1; act_2$	$\mathsf{Plan}_{.}(\mathsf{Pre}.([\![\varphi_1]\!] \mid \mathsf{PB.Seq}.([\![act_1]\!] \mid \mathsf{Cons}.[\![act_2]\!]))$
$Pl_2 = e_1 : \varphi_2 \leftarrow act_3; e_2; act_4$	$\mathsf{Plan}_{.}(\mathsf{Pre}.([\![\varphi_2]\!] \mid \mathsf{PB.Seq}.([\![act_3]\!] \mid \mathsf{Cons.Seq}.([\![e_2]\!] \mid \mathsf{Cons}.[\![act_4]\!])))$
$Pl_3 = e_2 : \varphi_3 \leftarrow act_5; act_6$	$\mathsf{Plan}_{.}(\mathsf{Pre}.([\![\varphi_3]\!] \mid \mathsf{PB.Seq}.([\![act_5]\!] \mid \mathsf{Cons}.[\![act_6]\!]))$
$\varphi_1 = b_1 \wedge b_2$	$\mathsf{B}(1) \mid \mathsf{B}(2)$
$act_1 = b_3 \leftarrow \langle \{b_4\}, \emptyset \rangle$	$\mathsf{Act}.(\mathsf{Pre.B}(3) \mid \mathsf{Add.B}(4) \mid \mathsf{Del}.1)$
$act_2 = b_4 \leftarrow \langle \{b_5\}, \emptyset \rangle$	$\mathsf{Act}.(\mathsf{Pre.B}(4) \mid \mathsf{Add.B}(5) \mid \mathsf{Del}.1)$
$\varphi_2 = b_6 \wedge b_7$	$\mathsf{B}(6) \mid \mathsf{B}(7)$
$act_3 = true \leftarrow \langle \{b_8\}, \emptyset \rangle$	$\mathsf{Act}.(\mathsf{Pre}.1 \mid \mathsf{Add.B}(8) \mid \mathsf{Del}.1)$
e_2	E_{e_2}
$act_4 = b_9 \leftarrow \langle \{b_5\}, \emptyset \} \rangle$	$\mathsf{Act}.(\mathsf{Pre.B}(9) \mid \mathsf{Add.B}(5) \mid \mathsf{Del}.1)$
$\varphi_3 = b_8$	$\mathsf{B}(8)$
$act_5 = b_8 \leftarrow \langle \{b_{10}\}, \emptyset \rangle$	$\mathsf{Act}.(\mathsf{Pre.B}(8) \mid \mathsf{Add.B}(10) \mid \mathsf{Del}.1)$
$act_6 = b_{10} \leftarrow \langle \{b_9\}, \{b_8, b_{10}\} \rangle$	$\mathsf{Act}.(\mathsf{Pre.B}(10) \mid \mathsf{Add.B}(9) \mid \mathsf{Del}.(\mathsf{B}(8) \mid \mathsf{B}(10)))$

lish the following equivalences to unify them under the same action encoding.

$$?\varphi \equiv act : \varphi \leftarrow \langle \emptyset, \emptyset \rangle \tag{21}$$

$$+b \equiv act : \emptyset \leftarrow \langle \{b\}, \emptyset \rangle \tag{22}$$

$$-b \equiv act : \emptyset \leftarrow \langle \emptyset, \{b\} \rangle \tag{23}$$

3.3. Example of Encoding

To show how our encoding works, Table 4 provides the mapping for a BDI agent for the travelling example in Table 1.

This completes the structural encoding (i.e. the syntactic specification of a BDI agent), we now turn our attention to a *behavioural* encoding of BDI agents (i.e. the operation semantics of a BDI agent) as a BRS. We do so in an incremental manner in the following three steps: in Section 4 we define the semantics of a subset of CAN that we call the core CAN. Core CAN semantics excludes concurrency and declarative goals, and so resembles AgentSpeak [2]. In Section 5 we encode core CAN as a BRS and in Section 6 we extend such a BRS for the core CAN to include concurrency and declarative goals.

4. Semantics of Core CAN Language

4.1. Overview of Core CAN *language*

The core operation of an agent in response to an (external) event is as follows. All relevant plans for that event are retrieved from the (pre-defined) plan library. An *applicable* plan is selected (if one exists) and its plan-body is added to the intention base. The plan-body consists of discrete steps, e.g. actions or sub-events. When executing a sub-event, its applicable plan requires to be found, and its plan-body is also added to the intention base – this forms an execution *tree* within the intention. A BDI agent continues to execute until there are no pending events, and all intentions are completed (either successfully or with failure).

4.2. Core CAN *Semantics*

We specify the behaviour of an agent as an operational semantics [27] defined over configurations $\mathcal{C}$ and transitions $\mathcal{C} \to \mathcal{C}'$. Transitions $\mathcal{C} \to \mathcal{C}'$ denote a single execution step between configuration $\mathcal{C}$ and $\mathcal{C}'$. We write $\mathcal{C} \to$ (resp. $\mathcal{C} \nrightarrow$) to state that there is some (resp. is not) $\mathcal{C}'$ such that $\mathcal{C} \to \mathcal{C}'$.

A derivation rule specifies the necessary conditions for an agent to transition to a new configuration. A derivation rule consists of a (possibly empty) set of premises p_i $(i = 1, \ldots, n)$ on $\mathcal{C}$, and a conclusion, denoted by

$$\frac{p_1 \quad p_2 \quad \cdots \quad p_n}{\mathcal{C} \to \mathcal{C}'} \quad l$$

where l is a rule name. We write $\mathcal{C} \xrightarrow{l} \mathcal{C}'$ to denote $\mathcal{C}$ evolves to $\mathcal{C}'$ through the application of derivation rule l.

The CAN semantics were originally defined [6] over the triple $\langle \mathcal{B}, \mathcal{A}, P \rangle$ where $\mathcal{B}$ is the current belief base, $\mathcal{A}$ the *sequence* of actions that have been executed, and P the current partially executed plan-body. As the recorded sequence of executed actions is never used to determine the operation of an agent, i.e. there are no pre-condition on $\mathcal{A}$, we do not include it here (i.e. $\langle \mathcal{B}, P \rangle$). It is trivial to log the action sequence within the bigraph model if required, however we do not do so here because an action log introduces states that would otherwise be isomorphic (resulting in larger transition systems).

The semantics of CAN language is specified by two types of transitions. The first transition type, denoted as $\to$, specifies *intention-level* evolution in terms of configuration $\langle \mathcal{B}, P \rangle$ where $\mathcal{B}$ is the current belief set, and P the plan-body currently being executed (i.e. the next step of the current intention). The second type, denoted as $\Rightarrow$, specifies *agent-level* evolution over $\langle E^e, \mathcal{B}, \Gamma \rangle$, detailing how to execute a complete agent where E^e stands for the a set of pending external events required to address.

Fig. 5 gives the set of derivation rules for evolving any single intention. For example, derivation rule *act* handles the execution of an action, when the pre-condition is met, resulting in a belief state update. Rules ?, $+b$ and $-b$ are special actions that perform pre-condition check (?), adding one belief atom ($+b$) and deleting atoms ($-b$). As in Section 3, we assume an equivalence between *act* and $?, +b, -b$ and do not directly model these rules. 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 (due to e.g. the unexpected environment changes). Finally, rules ; and $;_\top$ describe executing plan-bodies in sequence.

The agent-level semantics are given in Fig. 6. An agent configuration is defined by the triple $\langle E^e, \mathcal{B}, \Gamma \rangle$ consisting of a set of external events E^e to which the agent is required to respond, the belief set $\mathcal{B}$, and the intention base Γ – a set of partially executed plan-bodies P that the agent has already committed to. The derivation rule A_{event} handles external events, which originate from the

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

$$\frac{}{\langle \mathcal{B}, +b \rangle \rightarrow \langle \mathcal{B} \cup \{b\}, nil \rangle}\ +b \qquad \frac{}{\langle \mathcal{B}, -b \rangle \rightarrow \langle \mathcal{B} \setminus \{b\}, nil \rangle}\ -b$$

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

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

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

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

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

Figure 5: Core Can semantics.

$$\frac{e \in E^e}{\langle E^e, \mathcal{B}, \Gamma \rangle \Rightarrow \langle E^e \setminus \{e\}, \mathcal{B}, \Gamma \cup \{e\} \rangle} A_{event}$$

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

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

Figure 6: Derivation rules for agent configuration.

environment[4], by adopting them as intentions. Rule A_{step} selects an intention from the intention base, and evolves a single step w.r.t. intention-level transition, while A_{update} discards intentions which cannot make any intention-level transition (either because it has already succeeded, or it failed)[5].

4.3. Example of Core Can *Semantics*

To show how an agent evolves in the Can semantics we use the conference travelling example in Table 1. Assuming the external event e_1 has already been converted from a desire to an intention, Fig. 7 illustrates the intention-level evolution of this intention according to the rules presented in Fig. 5. In Fig. 7 agents evolve from left to right, each line consists of a single step of an intention. Below each step we show the sub-rules that applied. A commentary is as follows.

When the event e_1 is posted to the agent, the *event* rule in Fig. 5 transforms e_1 into the program containing all the relevant plans available (1). If the agent believes that it owns a car and the venue is within driving distance (i.e. φ_1 holds), then the *select* rule transforms the set of relevant plans into the selected plan (2), which indicates the sequence $act_1; act_2$ is ready for execution, while the other plans are indicated as backup on the right-hand side of the symbol $\rhd$. Next, the agent tries to execute the program $act_1; act_2$. Given the belief base in Table 1, the pre-condition act_1 does not hold (e.g. the car engine fails to start), thus $act_1 \not\rightarrow$. Meanwhile, the backup plan is applicable shown by the derivation *select* from $e_1 : (|\varphi_2 : act_3; e_2; act_4|)$ to $act_3; e_2; act_4 \rhd e_1 : (|\emptyset|)$. According to the rule $\rhd_\bot$, the agent can initiate the failure recovery by trying such a backup plan, resulting in the program shown in (3). Since act_3 has *true* as its pre-condition, it can always be executed shown by $act_3 \xrightarrow{act} nil$. After execution of act_3, the rule ; then updates the entire sequence from $act_3; e_2; act_4$ to $nil; e_2; act_4$. After the left-hand side of $\rhd$ is updated, the rule $\rhd_;$ can then further transform the program to that in (4). In order to discard the symbol *nil* in a sequence, it requires the part after *nil* in a sequence to be progressed, namely $e_2; act_4$. To progress the $e_2; act_4$, it requires to progress the first part of such a sequence, i.e. e_2. To progress the event e_2, it requires to retrieve a set of its relevant plans. Therefore, we have what is shown in the (5). The rule *select* firstly transforms the event e_2 to a set of relevant plans, secondly the rule ; updates the sequence $e_2; act_4$, and thirdly the symbol *nil* can be removed by the rule $;_\top$ from the entire sequence $nil; e_2; act_4$. Finally, the rule $\rhd_;$ can follow up transforming the entire program on the left-hand side of $\rhd$ accordingly.

For brevity, we omit the rest of the evolution. In practice an agent may execute multiple intentions concurrently.

---

[4]As we do not model the environment explicitly, we assume any events are waiting in the desire set at the start of an agent execution.

[5]In the original Can semantics there is no way to determine if an event was handled successfully or not, both cases are treated the same way (by removing the intention when it is done or cannot progress).

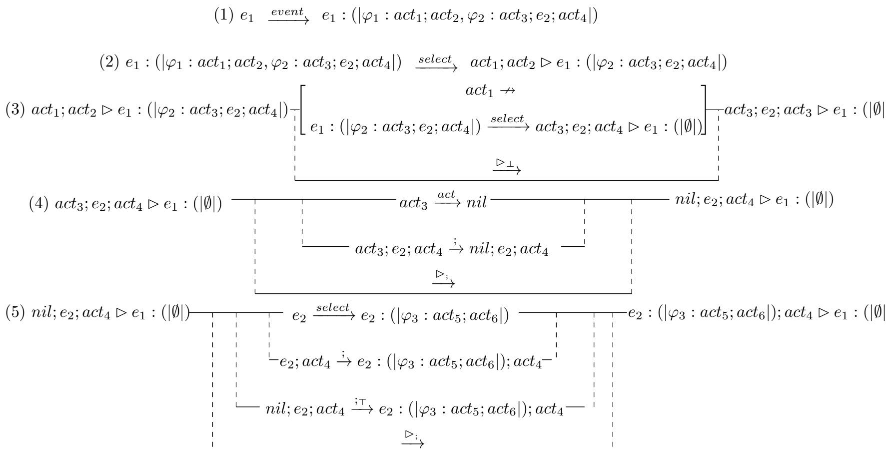


Figure 7: Illustration of intention-level evolution of the event e_1.

4.4. AND/OR Trees

We can view the semantic evolution of the agent program in terms of reductions over AND/OR trees, and use this representation to reason about the interactions between events, plans, and intentions [28, 29]. AND nodes are successful if **all** of their children succeed while OR nodes are successful if **at least one** child succeeds. We make heavy use of such AND/OR trees in our behavioural encoding of CAN semantics in bigraphs that can be seen (in part) as reductions over these trees. However, we stress that although the behaviour of a CAN agent can be visualised via AND/OR trees, in practice, the trees are not fully realised in memory and are created on-demand as the intention evolves.

The root of an AND/OR tree is a top-level external event represented as an OR node, that is, an event succeeds if at least one plan succeeds. The tree is built implicitly through the syntax of CAN. For example, the sequencing symbol ; ensures that execution must successfully execute all steps in the plan-body to allow the parent AND node to succeed. Meanwhile, the failure recovery symbol $\triangleright$ represents choice, with backup plans creating the branching structure. In Section 6.1, an additional form $\|$ will be introduced to complement ; by identifying branches that can be explored concurrently.

As an example we revisit the conference travelling example of Table 1 showing one possible AND/OR tree for the plans $Pl_1 = e_1 : \varphi_1 \leftarrow act_1; act_2$, $Pl_2 = e_1 : \varphi_2 \leftarrow act_3; e_2; act_4$, and $Pl_3 = e_2 : \varphi_3 \leftarrow act_5; act_6$. In this case, Pl_1 was chosen first and Pl_2 kept as a backup plan as shown in Fig. 8. The top-level event e_1 is achieved if either of the two plans Pl_1 or Pl_2 are successful. In this case the agent has chosen to do Pl_1 *before* Pl_2, although the ordering is not fixed ahead of execution time. The plan Pl_1 itself involves performing the actions act_1 *followed by* act_2, whereas one part of plan-body of plan Pl_2 involves achieving the sub-event e_2 which can, in turn, be addressed by the plan Pl_3.

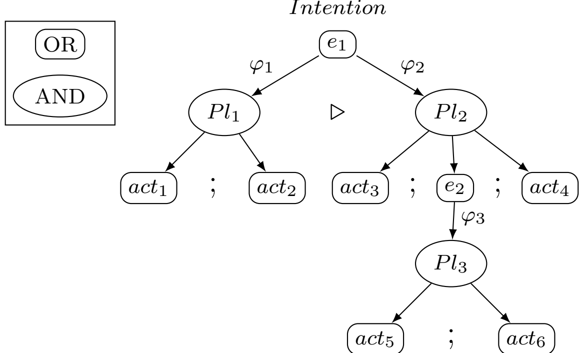


Figure 8: Snapshot of AND/OR tree representing the intention for the event e_1 where the agent chose to try Pl_1 before Pl_2 during execution.

From the point of view of the semantics, the tree is explored in a depth-first manner with reductions being pushed down the tree. For example, the derivation rule ; reduces a given branch of the tree while the rule $;_\top$ moves to the next child at the same AND level. When a node cannot be reduced, e.g. if an action pre-condition is unmet, this failure propagates to the closest branch point (OR-node) where they are handled by the failure recovery rules (e.g. $\triangleright_\bot$).

5. Encoding Core CAN Semantics in Bigraphs

We now encode the core CAN semantics (presented in Figs. 5 and 6) as a BRS and show that the encoding is faithful. By faithful we mean that for each transition $\xRightarrow{l}$ (resp. intention $\xrightarrow{l}$) $\langle E^e, \mathcal{B}, \Gamma\rangle \xRightarrow{l} \langle E'^e, \mathcal{B}', \Gamma'\rangle$ (resp. $\langle \mathcal{B}, P\rangle \xRightarrow{l} \langle \mathcal{B}', P'\rangle$) there exists a **finite** sequence of reaction rules, such that $[\![\langle E^e, \mathcal{B}, \Gamma\rangle]\!] \longrightarrow\!\!\!\!\triangleright^+ [\![\langle E'^e, \mathcal{B}', \Gamma'\rangle]\!]$ (resp. $[\![\langle \mathcal{B}, P\rangle]\!] \longrightarrow\!\!\!\!\triangleright^+ [\![\langle \mathcal{B}', P'\rangle]\!]$) and no new BDI derivation rule becomes applicable. The encoding may introduce new intermediate states, but there are no new applicable BDI derivation rules, i.e. there is no additional branching.

Throughout the remainder of the paper we use Eq. (9) from Fig. 3a to allow focusing on specific elements of an agent/intention, e.g. allowing us to ignore the plan library Π above as this is never mutated.

To encode control flow required for execution, we require additional entities that are not part of the structural encoding, i.e. they do not necessarily have a corresponding agent representation in CAN. These additional entities are given in Table 5 and their purpose is introduced as they are used.

For brevity, we give an overview of key aspects of the encoding. The full *executable* model, for use with BigraphER [19], is available [30].

5.1. Belief Checks and Updates

The CAN semantics assumes set operations and logic entailment as built-in operators. However, as we want an *executable* semantics, these must be explicitly encoded in the BRS.

Table 5: Additional entities for semantics encoding.

Description	Entity	Parent(s)	LinksTo	Diagrammatic Form
Set of beliefs to check	Check	Beliefs		
Unknown check result	CheckRes	{Act, Plan}		
Successful entailment	CheckRes.T	{Act, Plan}		
Failed entailment	CheckRes.F	{Act, Plan}		
Not-yet checked token	CheckToken	Plan		
Reduction of entity/site	Reduce	{Intention, Try, Seq, Conc, L, R}		
Reduction failure	ReduceF	{Intention, Seq, Try, L, R}		$\not\rightarrow$

(a) `check_T`(n)

(b) `check_end`

(c) `check_F`

Figure 9: Reactions for logical entailment.

We encode belief updates and checks in the usual recursive manner as shown in Fig. 9. For example, the reaction rule `check_end` provides a base-case for `check_T`(n), while `check_F` (a conditional rule) handles the case when there is no match in the belief base. Similar reaction rules (not shown) are provided to perform addition and deletion of beliefs[6].

The belief check reaction rules use auxiliary entities, e.g. Check_l and $\mathsf{CheckRes}_l$ (shown as a diamond). These auxiliary entities, which are added from other reaction rules, encode control flow. As these entities are not part of the Can syntax encoding, they do not enable any additional agent steps. After performing the sequence of reaction rules equivalent to a Can derivation rule, no auxiliary entities will be present – they are only allowed in intermediate states.

Notice the number of children of Check decreases on each reaction rule application suffices to prove that the logical entailment (resp. checks/updates) will complete in a *finite number* of steps. As such, placing belief checks/updates into the *highest* rule priority class of the BRS allows us to assume belief checks/updates are *atomic* with respect to the other reactions. That is, an agent never sees a part-modified belief set (as required to model atomic actions).

[6]We assume additions/deletions are disjoint (as they are in practice) so that there are no race conditions between the reactions.

We refer to the priority class of set operations with the label

$$\{\texttt{set_ops}\} \stackrel{\text{def}}{=} \{\texttt{check_T}(n), \texttt{check_end}, \texttt{check_F}, \texttt{del_in}(n), \texttt{del_notin}(n), \texttt{delete_end}, \texttt{add_end}, \texttt{add_notin}(n), \texttt{add_in}(n)\}$$

5.2. Modelling Reductions

The Can semantics assumes a notion of irreducibility (this is the same as negative premises in [31, 32]). That is, the derivation rule $\langle \mathcal{B}, P \rangle \nrightarrow$ represents the failure of an agent to perform any further operation on the program P under the belief $\mathcal{B}$, given all specified reducible rules in Can. For example, $\langle \mathcal{B}, act \rangle \nrightarrow$ holds if the pre-condition of the action *act* is not met.

While Can remains agnostic to such details, we require the notion of irreducibility to be encoded *explicitly* to obtain an executable semantics. To encode explicit reduction, we introduce auxiliary controls Reduce (by colouring the entity/site being reduced as red) and ReduceF (representing $\nrightarrow$).

Reduce requests the entities nested below are reduced, for example by executing an action. In the case reduction is not possible, e.g. if an action precondition is not met, ReduceF represents failure to reduce, enabling checks of the premise $\langle \mathcal{B}, P \rangle \nrightarrow$ in derivation rules.

If we view intentions as AND/OR trees, the explicit reductions perform the *tree search* with Reduce determining which sub-tree to reduce next, and ReduceF indicating a sub-tree could not reduce and backtracking should be performed.

We define a function $[\![\![\cdot]\!]\!] : \langle \mathcal{B}, P \rangle \rightarrow \mathbf{Bg}(\mathcal{K} \cup \mathsf{Reduce})$ that, for belief base $\mathcal{B}$ and intention-level program P, requests that the sub-tree rooted at P be reduced. That is:

$$[\![\![\langle \mathcal{B}, P \rangle]\!]\!] \stackrel{\text{def}}{=} [\![\mathcal{B}]\!] \parallel \mathsf{Reduce}.[\![P]\!]$$

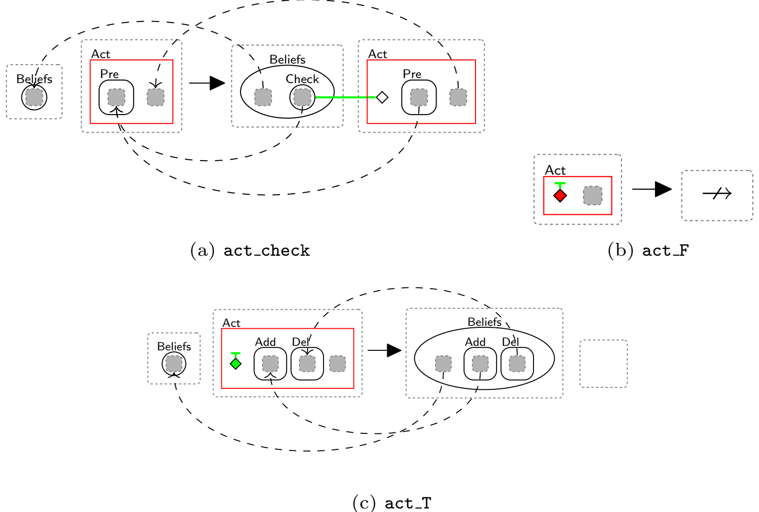
where $\mathcal{B}$ is a *mutable*, globally scoped, environment for the reduction of P. This is benefit of bigraphs for modelling: environments can be placed in parallel.

The function $[\![\![\cdot]\!]\!]$ for reduction plays a key role in our semantic encoding. For example, it forms the bridge between agent-level steps and intention-level steps, i.e. an agent $\langle \mathcal{B}, E^e, \{P \cup \Gamma\}, \Pi \rangle$ can (try to) step intention P using $[\![\![\langle \mathcal{B}, P \rangle]\!]\!]$.

5.3. Core Semantic Encoding

Given the atomic set operations and explicit reduction, we now show how the core Can semantics are encoded as a BRS. In the following lemmas, for readability, we allow free variables and assume the obvious interpretation. For example, we assume that P reduces to P', e_n is an event in E, etc.

Also, for readability, we refer to a "corresponding" reduction sequence to mean that there is a one to one correspondence between the Can step and the reduction sequence, no new possible reductions are introduced. The reduction sequence may not be unique, for example set operations can be performed in different orders, but the outcome of the sequence is the same as the Can step outcome, i.e. we model a big-step semantics through a sequence of small-steps.



(a) `act_check` (b) `act_F`

(c) `act_T`

Figure 10: Reactions for actions.

5.3.1. Actions

The main operation of an agent is to execute actions that update both the external environment, e.g. moving a block, and in-turn revise the internal belief base. Recall that, in the encoding of syntax of CAN, we have established entailment and belief state updates (rules $?\varphi, +b, -b$) as special cases of actions that simply do not update the external environment. As such, we can safely omit the explicit reactions for entailment and belief state updates.

If the pre-condition of an action is true, i.e. $\mathcal{B} \models \phi$, performing (or reducing) an action consists of the reactions `act_check` and `act_T` as shown in Fig. 10. Firstly, the reaction `act_check` requests the action pre-condition to be checked by nesting a Check entity within the belief base. As we have established set operations to be the highest priority class, we know a belief check operation is finite and applies atomically. Therefore, it does not alter the shape of $[\![\mathcal{B}]\!]$ (i.e. no other CAN rules are enabled). After successful entailment of the action pre-condition, the reaction rule `act_T` performs the action by updating the belief base. Once again, given the priority of set operations, the set updates will be effectively atomic and no other CAN derivation rule can interrupt such a belief update.

Lemma 1. *(Faithfulness of act) act has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, act : \varphi \leftarrow \langle \phi^+, \phi^- \rangle \rangle]\!] \longrightarrow\!\!\triangleright^+ [\![\langle \mathcal{B}', nil \rangle]\!]$.

Proof. We show $\longrightarrow\!\!\triangleright^+$ has form $\xrightarrow{\texttt{act_check}}\!\!\triangleright \xrightarrow{\{\texttt{set_ops}\}}\!\!\triangleright^* \xrightarrow{\texttt{act_T}}\!\!\triangleright \xrightarrow{\{\texttt{set_ops}\}}\!\!\triangleright^*$ and intermediate states do not allow additional branching. Consider the transitions applicable in each of the four steps; recall the rule priority is `act_check` $<$ `act_T` $< \{$`set_ops`$\}$.

Step 1 $\xrightarrow{\texttt{act_check}}\!\!\triangleright$**.** The initial state $[\![\langle \mathcal{B}, act : \varphi \leftarrow \langle \phi^+, \phi^- \rangle \rangle]\!]$ is

$\mathsf{Beliefs}.([\![b_1]\!] \mid \ldots \mid [\![b_n]\!]) \parallel \mathsf{Reduce}.\mathsf{Act}.(\mathsf{Pre}.[\![\varphi]\!] \mid \mathsf{Add}.[\![\phi^+]\!] \mid \mathsf{Del}.[\![\phi^-]\!])$.

No transition in $\xrightarrow{\{\texttt{set_ops}\}}\!\!\triangleright$ is applicable because Beliefs contains only $\mathsf{B}(n)$ entities and Act does not contains Check_l entities. Similarly, $\xrightarrow{\texttt{act_T}}\!\!\triangleright$ is not applicable. $\xrightarrow{\texttt{act_check}}\!\!\triangleright$ is applicable (lhs Fig. 10a), resulting in a new Check_l in Beliefs (rhs Fig. 10a). No other transitions are applicable.

Step 2 $\xrightarrow{\{\texttt{set_ops}\}}\!\!\triangleright^*$**.** We now have state

$\mathsf{Beliefs}.(([\![b_1]\!] \mid \ldots \mid [\![b_n]\!]) \mid \mathsf{Check}_l.[\![\varphi]\!]) \parallel \mathsf{Reduce}.\mathsf{Act}.(\mathsf{Pre}.[\![\varphi]\!] \mid \mathsf{Add}.[\![\phi^+]\!] \mid \mathsf{Del}.[\![\phi^-]\!] \mid \mathsf{CheckRes}_l.1)$

Transitions in $\xrightarrow{\{\texttt{set_ops}\}}\!\!\triangleright^*$ are applicable and there are three cases to consider (induction and 2 base cases: lhs of Figs. 9a, 9b and 9c). No other transitions are applicable. Together, these transitions reduce in size the number of beliefs to be checked or remove Check_l, resulting in a finite sequence. In detail:

Case $\texttt{check_T}(n)$**.** There is at least one $\mathsf{B}(n)$ in Check_l, and a matching $\mathsf{B}(n)$ in $[\![b_1]\!] \mid \ldots \mid [\![b_n]\!]$ and so $\xrightarrow{\texttt{check_T(n)}}\!\!\triangleright$ applies, which reduces the number of children of Check_l by 1.

Case $\texttt{check_F}$**.** There is at least one $\mathsf{B}(n)$ in Check_l but no match in $[\![b_1]\!] \mid \ldots \mid [\![b_n]\!]$. This case cannot occur because the *act* precondition φ holds.

Case $\texttt{check_end}$**.** Check_l is empty, in which case $\xrightarrow{\texttt{check_end}}\!\!\triangleright$ applies, resulting in the removal of Check_l.

Step 3 $\xrightarrow{\texttt{act_T}}\!\!\triangleright$**.** We now have state

$\mathsf{Beliefs}.([\![b_1]\!] \mid \ldots \mid [\![b_n]\!]) \parallel \mathsf{Reduce}.\mathsf{Act}.(\mathsf{Pre}.[\![\varphi]\!] \mid \mathsf{Add}.[\![\phi^+]\!] \mid \mathsf{Del}.[\![\phi^-]\!] \mid \mathsf{CheckRes}_l.\mathsf{T})$.

Transitions in $\xrightarrow{\{\texttt{set_ops}\}}\!\!\triangleright$ are not applicable, but $\xrightarrow{\texttt{act_T}}\!\!\triangleright$ is applicable, since $\mathsf{CheckRes}_l.\mathsf{T}$ is present and the precondition holds.

Step 4 $\xrightarrow{\{\texttt{set_ops}\}}\!\!\triangleright$**.** We now have state

$\mathsf{Beliefs}.(([\![b_1]\!] \mid \ldots \mid [\![b_n]\!]) \mid \mathsf{Add}.[\![\phi^+]\!] \mid \mathsf{Del}.[\![\phi^-]\!]) \parallel 1$.

Similar to step 2, only transitions in $\xrightarrow{\{\texttt{set_ops}\}}\!\!\triangleright$ apply, in this case a finite number of times until Add and Del are removed. No new branching is introduced, and we are left with bigraph $\mathsf{Beliefs}.([\![b_i]\!] \mid \ldots \mid [\![b_j]\!]) \parallel 1$, as required, (recall $[\![nil]\!] \stackrel{\text{def}}{=} 1$). $\mathcal{B}' = [\![b_i]\!] \mid \ldots \mid [\![b_j]\!]$ is $\mathcal{B}$ with the belief additions/deletions performed.

□

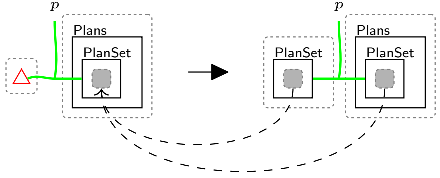


Figure 11: `reduce_event`.

We have discussed what happens when the pre-condition of an action holds. However, it is not explicit in CAN semantics what should be done in the case the pre-condition check fails. From the perspective of an AND/OR tree, as actions are always under AND nodes, the failure needs to be propagated upwards in order to enable failure recovery to take place. As introduced in Section 5.2, the entity ReduceF is provided to denote explicitly the reduction failure. Therefore, we have the reaction `act_F` to report the failure, given in Fig. 10b. Reducing to ReduceF enables checking of the premise $\langle B, act\rangle \nrightarrow$ (as is done implicitly in CAN semantics). Once a failure is reported, other reaction rules can be triggered to, for example, recover from the failure.

5.3.2. Plan Selection

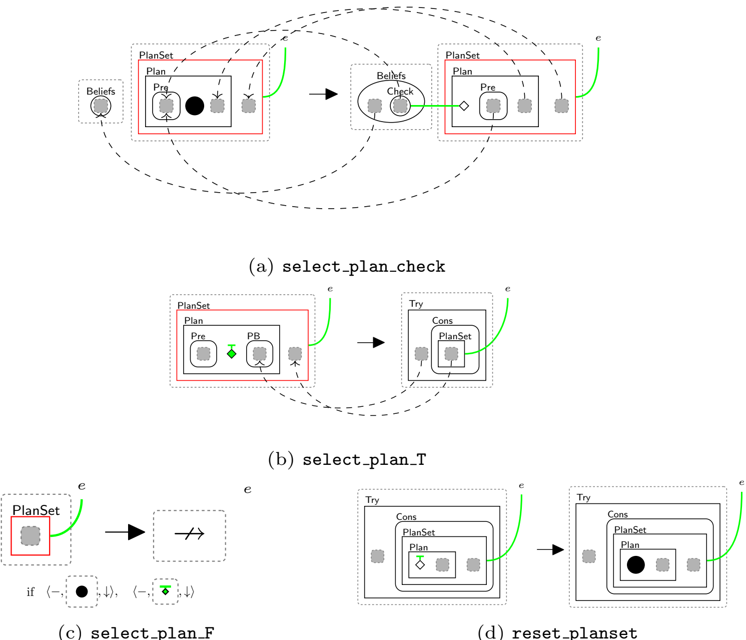
Recall that the agent responds to an event by selecting an applicable plan from a set of pre-defined plans. The following two derivation rules specify the plan selection. The first rule *event* converts an event to the set of plans that respond to that event (i.e. relevant plans), while the second rule *select* chooses an applicable plan (if exists) from the set of relevant plans.

The reaction rule corresponding to the derivation rule *event* is depicted in Fig. 11. As the syntax encoding uses links to connect an event E_e to its set of relevant plans $\mathsf{PlanSet}_e$, we can encode the derivation rule *event* with a single reaction rule by replacing the event entity E_e with $\mathsf{PlanSet}_e$ as shown in Fig. 11.

Lemma 2. *(Faithfulness of event) event has a corresponding finite reaction sequence* $[\![\langle\mathcal{B}, e\rangle]\!] \longrightarrow\!\!\!\!\triangleright^{+} [\![\langle\mathcal{B}, e : (\!|\ \Delta\ |\!)\rangle]\!]$.

Proof. $\longrightarrow\!\!\!\!\triangleright^{+}$ corresponds to $\xrightarrow{\texttt{reduce_event}}\!\!\!\!\triangleright$. Trivial. □

The derivation rule *select* is modelled in a similar style to how to execute an action, beginning with the pre-condition check against the belief base before selecting an appropriate plan (if one exists). In detail, the reaction rule `select_plan_check` (in Fig. 12a) finds a plan that has not yet had the pre-conditions checked—facilitated via an automatically-added auxiliary entity CheckToken that records if a plan has already been considered—and initiates an operation to check the plan pre-condition. To ensure the automatic addition of the entity CheckToken to all Plan entities within the Plans perspective, an additional reaction rule is executed once at the start of a model execution to update the plan library. This is an implementation detail, we do not add the



(a) `select_plan_check`

(b) `select_plan_T`

(c) `select_plan_F`

(d) `reset_planset`

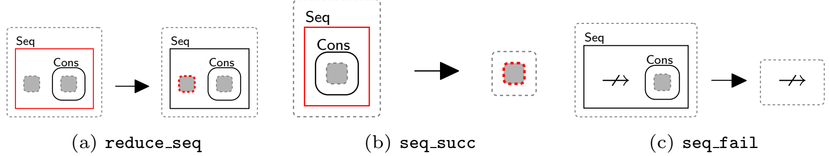
Figure 12: Reactions for plan selection.

tokens directly to the syntax encoding in the section Section 3. After checking the pre-condition of a plan is true, the reaction rule `select_plan_T` (in Fig. 12b) removes the selected applicable plan from the set of relevant plans, and converts it into $\rhd$ form, keeping the rest of plans as backups.

Lemma 3. *(Faithfulness of select) When the set of relevant plans* $(|\Delta|)$ *is non-empty and contains* at least *one applicable plan for a given event, select has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, e : (|\Delta|)\rangle]\!] \longrightarrow\!\!\rhd^{+} [\![\langle \mathcal{B}, P \rhd e : (\Delta \setminus \{\varphi : P\})\rangle]\!]$.

Proof. Let $\xrightarrow{\mathit{select_and_check}}\!\!\rhd \; = \; \xrightarrow{\texttt{select_plan_check}}\!\!\rhd \; \xrightarrow{\texttt{\{set_ops\}}}\!\!\rhd^{*}$ be the sequence of rules that selects and tests the pre-condition for a plan. $\longrightarrow\!\!\rhd^{+}$ has form $\xrightarrow{\mathit{select_and_check}}\!\!\rhd^{+} \; \xrightarrow{\texttt{select_plan_T}}\!\!\rhd$. We assume at least one applicable plan. The proof is similar to Lemma 1; in this case the transitions in $\xrightarrow{\mathit{select_and_check}}\!\!\rhd$ reduce the number of plans to be checked until an applicable plan is selected. Transitions in $\xrightarrow{\mathit{select_and_check}}\!\!\rhd$ only introduce auxiliary controls, so no other transitions are enabled. □

If no plan is applicable (a failure), the reaction rule `select_plan_F` (in Fig. 12c) propagates a ReduceF up the tree. We use a *conditional* rule to ensure the plan selection only fails if *all* plans have been checked (or there are no plans), i.e. when



(a) `reduce_seq` (b) `seq_succ` (c) `seq_fail`

Figure 13: Reactions for sequencing with priorities: `reduce_seq` $<$ {`seq_succ`, `sec_fail`}

there are no CheckToken entities left, and no plan that was checked is applicable. Finally, an auxiliary reaction rule `reset_planset` (Fig. 12d) ensures that after plan selection, the remaining unchosen (but checked) plans are re-assigned the control CheckToken to allow the plan to be checked again if failure recovery is required.

5.3.3. Tree Reductions

The remaining CAN intention-level derivation rules specify how the AND/OR tree should be explored.

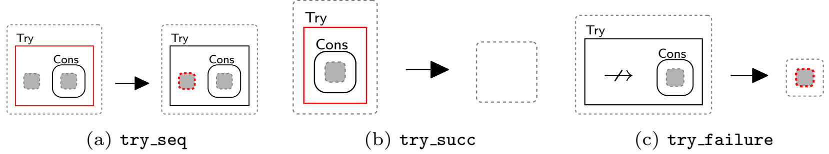
The derivation rules ; and $;_\top$ describe how to progress the sequencing of $P_1; P_2$. The derivation rule ; is encoded by the reaction rule `reduce_seq` (Fig. 13a) that pushes reduction into the first child of a sequence. The use of a site (i.e. an abstraction) in bigraphs allows this single rule to handle any type of program P. In more detail, `reduce_seq` is a *generalisation* of `seq_succ` and `seq_fail`. For example, if we get remove id under the control Seq (not id under Cons) on the left-hand side of reaction rule `reduce_seq`, we get the reaction rule `seq_succ`. Therefore, we enforce a priority ordering on the reaction rules as given in Fig. 13 to ensure that the special cases are applied only when needed.

Lemma 4. *(Faithfulness of ;) ; has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, P_1; P_2 \rangle]\!] \rightarrowtriangle^+ [\![\langle \mathcal{B}', P_1'; P_2 \rangle]\!]$.

Proof. The initial state is $[\![\mathcal{B}]\!] \parallel \mathsf{Reduce}.(\mathsf{Seq}.[\![P_1]\!] \mid \mathsf{Cons}.[\![P_2]\!])$. Assume $\langle \mathcal{B}, P_1 \rangle \to \langle \mathcal{B}', P_1' \rangle$. P_1 can reduce so it cannot be *nil*, thus the rule `reduce_seq` applies resulting in bigraph $[\![\mathcal{B}]\!] \parallel (\mathsf{Seq}.\mathsf{Reduce}.[\![P_1]\!] \mid \mathsf{Cons}.[\![P_2]\!])$, which matches bigraph $[\![\langle \mathcal{B}, P_1 \rangle]\!] = [\![\mathcal{B}]\!] \parallel \mathsf{Reduce}.[\![P_1]\!]$. P_1 is then reduced to P_1', with beliefs $\mathcal{B}'$, (using assumption) and the result is state $[\![\mathcal{B}']\!] \parallel (\mathsf{Seq}.[\![P_1']\!] \mid \mathsf{Cons}.[\![P_2]\!])$, which is equivalent to $[\![\langle \mathcal{B}', P_1'; P_2 \rangle]\!]$. No other transitions are possible. □

The derivation rule $;_\top$ is encoded using the reaction rule `seq_succ` (Fig. 13) that matches in the case the first part of the sequence completed successfully, i.e. $[\![nil]\!] = 1$. As specified in the derivation rule in CAN, we not only make the children under Cons the new current program, but we also (try to) reduce it immediately.

Lemma 5. *(Faithfulness of* $;_\top$*)* $;_\top$ *has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, nil; P_2 \rangle]\!] \rightarrowtriangle^+ [\![\langle \mathcal{B}', P_2' \rangle]\!]$.



(a) try_seq (b) try_succ (c) try_failure

Figure 14: Reactions for recovery with priorities: try_seq < {try_succ, try_failure}.

Proof. The initial state is $[\![\mathcal{B}]\!] \parallel \mathsf{Reduce}.(\mathsf{Seq}.[\![nil]\!] \mid \mathsf{Cons}.[\![P_2]\!])$. Assume $\langle \mathcal{B}, P_2\rangle \to \langle \mathcal{B}', P_2'\rangle$. The rule seq_succ applies, resulting in $[\![\langle \mathcal{B}, P_2\rangle]\!]$, which, by the assumption, is reduced to P_2', with beliefs $\mathcal{B}'$, i.e. $[\![\langle \mathcal{B}', P_2'\rangle]\!]$. No other transitions are possible. □

5.3.4. Failure Recovery

If we cannot reduce a sequence, then a failure is propagated up the tree through the reaction rule seq_fail (Fig. 13c). It is important that the reaction rule seq_fail, and later failure cases, do not require the left-hand entity to be under a Reduce. This means a reaction can be applied as soon as a failure is discovered, rather than the *next* time the agent attempts to advance the intention. This matches the CAN semantics that handle failure of intention immediately ($\rhd_\bot$ in Fig. 5). If no backup plans apply e.g. there are no plans left to select, the failure is pushed upwards through the reaction rule select_plan_F (Fig. 12c). in this case, the transition labelled reaction rule try_failure does not apply as the Cons entity has been removed. As with sequencing, a priority order is required since try_seq generalises the other reactions.

We now consider the derivation rules $\rhd_;$, $\rhd_\top$, and $\rhd_\bot$ that relate to failure recovery. The reaction rule try_seq (Fig. 14a) encodes the derivation rule $\rhd_;$ by pushing reduction into the left hand side of the $\rhd$ operator, if no failure occurs.

Lemma 6. *(Faithfulness of $\rhd_;$) $\rhd_;$ has a corresponding finite reaction sequence $[\![\langle \mathcal{B}, P_1 \rhd P_2\rangle]\!] \longrightarrow\!\!\rhd^+ [\![\langle \mathcal{B}', P_1' \rhd P_2\rangle]\!]$.*

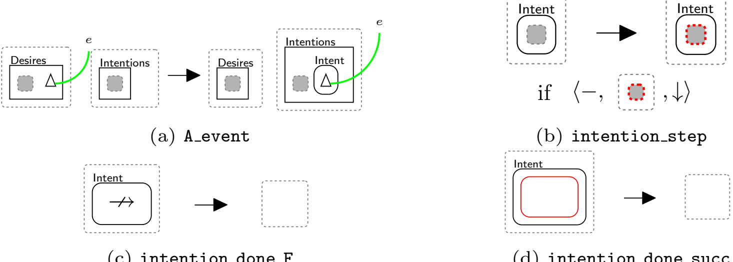
Proof. The argument is similar to Lemma 4, starting from initial state $[\![\mathcal{B}]\!] \parallel \mathsf{Reduce}.(\mathsf{Try}.[\![P_1]\!] \mid \mathsf{Cons}.[\![P_2]\!])$ and transition from Fig. 14a. □

If the selected plan was executed successfully, the reaction rule try_succ (in Fig. 14b) encodes the derivation rule $\rhd_\top$ to propagate *success* up-the-tree by removing the $\rhd$ structure.

Lemma 7. *(Faithfulness of $\rhd_\top$) $\rhd_\top$, has a corresponding finite reaction sequence $[\![\langle \mathcal{B}, nil \rhd P_2\rangle]\!] \longrightarrow\!\!\rhd^+ [\![\langle \mathcal{B}, nil\rangle]\!]$.*

Proof. $\longrightarrow\!\!\rhd^+$ corresponds to $\xrightarrow{\texttt{try_succ}}\!\!\rhd$. Trivial. □

Finally, the transition labelled try_failure (Fig. 14c) encodes the derivation rule $\rhd_\bot$. This is the first instance where ReduceF is used as a *premise* to denote a program that failed to progress. To recover, the failed program is deleted and



(a) A_event (b) intention_step

(c) intention_done_F (d) intention_done_succ

Figure 15: Agent level reactions with priorities: {A_event, intention_step} < {intention_done_F, intention_done_succ}.

the agent tries to reduce the right-hand side of $\rhd$ (i.e. by choosing from the remaining the set of relevant plans).

Lemma 8. *(Faithfulness of $\rhd_\bot$) $\rhd_\bot$ has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, P_1 \rhd P_2 \rangle]\!] \longrightarrow\!\!\rhd^+ [\![\langle \mathcal{B}', P_2' \rangle]\!]$ *when* $\langle B, P_1 \rangle \not\to$.

Proof. Assume $\langle \mathcal{B}, P_1 \rangle \not\to$, and $\langle \mathcal{B}, P_2 \rangle \to \langle \mathcal{B}', P_2' \rangle$. The initial state is $[\![\mathcal{B}]\!] \parallel \mathsf{Reduce.Try.}([\![P_1]\!] \mid \mathsf{Cons.}[\![P_2]\!])$. Rule `try_seq` (Fig. 14a) applies (as in Lemma 6), however, since $\langle \mathcal{B}, P_1 \rangle \not\to$, P_1 must reduce to $\mathsf{ReduceF}$. Rule `try_failure` then applies, resulting in a bigraph matching $[\![\langle \mathcal{B}, P_2 \rangle]\!]$. Through the second assumption, this is reduced to P_2', with beliefs $\mathcal{B}'$. No other transitions are possible.

□

5.3.5. Agent Steps

To complete the core semantics of Can, we now encode the agent-level derivation rules: A_{event}, A_{step}, and A_{update}.

The derivation rule A_{event} allows the agent to respond to an external event by adopting it in the intention base. This is encoded by reaction rule `A_event` (Fig. 15a) that simply moves the event from a desire to an intention.

Lemma 9. *(Faithfulness of A_{event}) A_{event} has a corresponding finite reaction sequence* $[\![\langle E^e \cup \{e_n\}, \mathcal{B}, \Gamma \rangle]\!] \longrightarrow\!\!\rhd^+ [\![\langle E^e, \mathcal{B}', \Gamma \cup \{e_n\} \rangle]\!]$.

Proof. $\longrightarrow\!\!\rhd^+$ corresponds to $\xrightarrow{\texttt{A_event}}\!\!\rhd$. Trivial. □

The derivation rule A_{step} allows the agent to execute a given intention by one reduction step. This is encoded with reaction rule `intention_step` (in Fig. 15b) that pushes a reduction into an intention (down the tree) if it is not already being reduced. This rule *introduces* the reduction form $[\![\cdot]\!]$ to an intention.

If the reduction is successful, we are left with a new updated P' (and $\mathcal{B}'$) as required. Unlike the Can derivation rule that removes the old intention and replaces it with a modified intention, ours is updated in-place. Multiple intentions can be reduced concurrently, e.g. `intention_step` can be applied to two different intentions in an interleaved fashion.

Lemma 10. *(Faithfulness of A_{step}) A_{step} has a corresponding finite reaction sequence* $[\![\langle E^e, \mathcal{B}, \Gamma \cup \{P\}\rangle]\!] \longrightarrow\!\triangleright^+ [\![\langle E^e, \mathcal{B}', \Gamma \cup \{P'\}\rangle]\!]$ *when* $\langle \mathcal{B}, P\rangle \rightarrow \langle \mathcal{B}', P'\rangle$.

Proof. Assume $[\![\langle \mathcal{B}, P\rangle]\!] \longrightarrow\!\triangleright^+ [\![\langle \mathcal{B}', P'\rangle]\!]$ and no intention is currently being reduced. By rule `intention_step` (Fig. 15b), P transitions to Intent.Reduce.P. This matches $[\![\langle \mathcal{B}, P\rangle]\!]$, which by assumption, reduces to $[\![\langle \mathcal{B}', P'\rangle]\!]$. This gives Intent.$[\![P']\!]$, with beliefs $\mathcal{B}'$. No other transitions are possible.

□

The derivation rule A_{update} is encoded by reaction rules `intention_done_F` (Fig. 15c) and `intention_done_succ` (Fig. 15d). The reaction rule `intention_done_F` handles the case where there was a failure to progress an intention. That is, if after pushing a reduction into the intention (via `intention_step`), we eventually consider Intent.ReduceF. The reaction rule `intention_done_succ` is a special case of `intention_done_F` for when a intention completed successfully (Intent.1). As the A_{update} rule only applies on *failure* to reduce an intention, `intention_done_succ` matches the form where we have tried to reduce an intention with the *nil* program inside. Importantly, this means that if an intention finishes an execution with $P = nil$, it is not until the *next* attempt to reduce it that A_{update} is applied. This mirrors the Can semantics that cannot tell if an intention is removed because it is finished, or if it failed.

Lemma 11. *(Faithfulness of A_{update}) A_{update} has a corresponding finite reaction sequence* $[\![\langle E^e, \mathcal{B}, \Gamma \cup \{P\}\rangle]\!] \longrightarrow\!\triangleright^+ [\![\langle E^e, \mathcal{B}', \Gamma\rangle]\!]$ *when* $\langle \mathcal{B}, P\rangle \not\rightarrow$.

Proof. Two cases.

Case 1. If $\langle \mathcal{B}, P\rangle \not\rightarrow$, then P must eventually reduce to a state that allows Intent.ReduceF to be matched, and $\xrightarrow{\texttt{intention_done_F}}\!\triangleright$ applies, completing the reaction sequence.

Case 2. If $P = nil$, then $\longrightarrow\!\triangleright^+$ corresponds to $\xrightarrow{\texttt{intention_step}}\!\triangleright \xrightarrow{\texttt{intention_done_succ}}\!\triangleright$. No rule can reduce Reduce.1 further.

□

To ensure agent-level transitions apply only when there is no intention currently being reduced (i.e. no intention-level transitions are being applied), both `intention_step` and `A_event` are in the lowest priority class. As `intention_step` is a *generalisation* of `intention_done_F` and `intention_done_succ`, the latter two have higher priority class than `intention_step` (and `A_event`).

5.4. Correctness

We can now give the main theorem, which states that the Can derivation rules can be encoded by a corresponding finite sequence of reaction rules.

Theorem 1 (Faithfulness)**.** *For each* Can *step* $\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E'^e, \mathcal{B}', \Gamma'\rangle$ *there exists a corresponding finite sequence of reactions such that*

$$[\![\langle E^e, \mathcal{B}, \Gamma\rangle]\!] \longrightarrow\!\triangleright^+ [\![\langle E'^e, \mathcal{B}', \Gamma'\rangle]\!] \; .$$

Proof. Follows from Lemmas 1–11 above. □

5.5. Reduction Example

To show how reduction works, in particular how failures are propagated through the AND/OR tree, we re-visit our running conference travelling example to address external event e_1. Consider the following configuration:

$$Agent = \langle \mathcal{B} = \{b_1, b_2, b_6, b_7\}, P = e_1 \rangle$$

This configuration has the current belief base $\mathcal{B}$ and current intention $P = e_1$. The bigraph (omitting desires and plan library) is:

$$[\![Agent]\!] = \mathsf{Beliefs.(B(1) \mid B(2) \mid B(6) \mid B(7))} \parallel \mathsf{Intent.E}_{e_1}$$

The detailed reduction step is given in Fig. 16. For succinctness, whenever appropriate, we use the mapping function to denote the part of bigraph encoding e.g. $[\![Pl_2]\!]$ while keeping the belief base implicitly as the background. The top-side of the reaction rule indicates the reaction rule that is applied and the bottom-side of the reaction rule indicates the result of application of the reaction rule, with line number in the beginning of each line. A short commentary is as follows. In line (1), the agent starts with an event to address. The reaction rule `intention_step` introduces the entity Reduce. Lines (2) and (3) show that to reduce an event, the event is replaced with its relevant plans. Reaction rule `intention_step` once again introduces Reduce for selection of an applicable plan. Lines (4) to (6) shows the successful selection of an applicable plan, plan Pl_1. From line (8) to line (9), the reaction rule `try_seq` pushes reduction in the left-hand side of the $\triangleright$ symbol, and from line (9) to line (10), the reaction rule `reduce_seq` pushes the reduction into the first child of a sequence. Lines (10) to (12) shows the execution of an action. In this case, we can see that the pre-condition of the action is not met, thus producing the entity ReduceF. As a consequence, this triggers failure recovery by deleting the failed program. Finally, lines (13) to (16) provides the successful re-selection of another applicable plan, namely plan Pl_2.

6. Extended Features

The full Can language also supports *concurrency* within plan-bodies, and *declarative goals* which allow an event to be repeatedly pursued until specified success/failure conditions holds. We now show how these features are encoded as bigraph reaction rules.

6.1. Concurrency

The Can semantics for concurrency are given in Fig. 17. They allow two branches within a single AND/OR tree to be reduced concurrently. For example, concurrency allows an agent to pursue two sub-tasks (i.e. two sub-events) but the ordering does not matter as long as they are all achieved eventually. The concurrency construct does not allow true concurrency, e.g. two branching reducing at the exact same time (same agent step), instead one of the branches is chosen and reduced at agent each step, i.e. in an interleaving manner. An

(1) $\mathsf{Intent.E}_{e_1}$

$\xrightarrow{\texttt{intention_step}}$

(2) $\mathsf{Intent.Reduce.E}_{e_1}$

$\xrightarrow{\texttt{reduce_event}}$

(3) $\mathsf{Intent.PlanSet}_{e_1}\mathsf{.(Plan_{.}(Pre.}[\![\varphi_1]\!] \mid \mathsf{PB.Seq.(}[\![act_1]\!] \mid \mathsf{Cons.}[\![act_2]\!])) \mid [\![Pl_2]\!])$

$\xrightarrow{\texttt{intention_step}}$

(4) $\mathsf{Intent.Reduce.PlanSet}_{e_1}\mathsf{.(Plan_{.}(Pre.}[\![\varphi_1]\!] \mid \mathsf{PB.Seq.(}[\![act_1]\!] \mid \mathsf{Cons.}[\![act_2]\!])) \mid [\![Pl_2]\!])$

$\xrightarrow{\texttt{select_plan_check}}$

(5) $\mathsf{Intent.Reduce.PlanSet}_{e_1}\mathsf{.(Plan_{.}(CheckRes.1} \mid \mathsf{Pre.}[\![\varphi_1]\!] \mid \mathsf{PB.Seq.(}[\![act_1]\!] \mid \mathsf{Cons.}[\![act_2]\!])) \mid [\![Pl_2]\!])$

$\xrightarrow{\texttt{set_ops}}{}^{*}$

(6) $\mathsf{Intent.Reduce.PlanSet}_{e_1}\mathsf{.(Plan_{.}(CheckRes.T} \mid \mathsf{Pre.}[\![\varphi_1]\!] \mid \mathsf{PB.Seq.(}[\![act_1]\!] \mid \mathsf{Cons.}[\![act_2]\!])) \mid [\![Pl_2]\!])$

$\xrightarrow{\texttt{select_plan_T}}$

(7) $\mathsf{Intent.Try.(Seq.(}[\![act_1]\!] \mid \mathsf{Cons.}[\![act_2]\!]) \mid \mathsf{Cons.PlanSet}_{e_1}.[\![Pl_2]\!])$

$\xrightarrow{\texttt{intention_step}}$

(8) $\mathsf{Intent.Reduce.Try.(Seq.(}[\![act_1]\!] \mid \mathsf{Cons.}[\![act_2]\!]) \mid \mathsf{Cons.PlanSet}_{e_1}.[\![Pl_2]\!])$

$\xrightarrow{\texttt{try_seq}}$

(9) $\mathsf{Intent.Try.(Reduce.Seq.(}[\![act_1]\!] \mid \mathsf{Cons.}[\![act_2]\!]) \mid \mathsf{Cons.PlanSet}_{e_1}.[\![Pl_2]\!])$

$\xrightarrow{\texttt{reduce_seq}}$

(10) $\mathsf{Intent.Try.(Seq.(Reduce.Act.(Pre.B(3)} \mid \mathsf{Add.B(4)} \mid \mathsf{Del.1)} \mid \mathsf{Cons.}[\![act_2]\!]) \mid \mathsf{Cons.PlanSet}_{e_1}.[\![Pl_2]\!])$

$\xrightarrow{\texttt{act_check}}$

(11) $\mathsf{Intent.Try.(Seq.(Reduce.Act.(CheckRes.1} \mid \mathsf{Pre.B(3)} \mid \mathsf{Add.B(4)} \mid \mathsf{Del.1)} \mid \mathsf{Cons.}[\![act_2]\!]) \mid \mathsf{Cons.PlanSet}_{e_1}.[\![Pl_2]\!])$

$\xrightarrow{\texttt{set_ops}}{}^{*}$

(12) $\mathsf{Intent.Try.(Seq.(Reduce.Act.(CheckRes.F} \mid \mathsf{Pre.B(3)} \mid \mathsf{Add.B(4)} \mid \mathsf{Del.1)} \mid \mathsf{Cons.}[\![act_2]\!]) \mid \mathsf{Cons.PlanSet}_{e_1}.[\![Pl_2]\!])$

$\xrightarrow{\texttt{act_F}}$ (as $\mathcal{B} \not\models b_3$)

(13) $\mathsf{Intent.Try.(ReduceF} \mid \mathsf{Cons.PlanSet}_{e_1}.[\![Pl_2]\!]))$

$\xrightarrow{\texttt{try_failure}}$

(14) $\mathsf{Intent.Reduce.PlanSet}_{e_1}.[\![Pl_2]\!]$

$\xrightarrow{\texttt{select_plan_check}}$

(15) $\mathsf{Intent.Reduce.PlanSet}_{e_1}\mathsf{.Plan_{.}(CheckRes.1} \mid \mathsf{Pre.}[\![\varphi_2]\!] \mid \mathsf{PB.Seq.(}[\![act_3]\!] \mid \mathsf{Cons.Seq.(}[\![e_2]\!] \mid \mathsf{Cons.}[\![act_4]\!])))$

$\xrightarrow{\texttt{select_plan_T}}$

(16) $\mathsf{Intent.Try.(Seq.(}[\![act_3]\!] \mid \mathsf{Cons.Seq.(}[\![e_2]\!] \mid \mathsf{Cons.}[\![act_4]\!])) \mid \mathsf{Cons.PlanSet}_{e_1}\mathsf{.1)}$

Figure 16: Example bigraph reduction of event e_1 from Table 1.

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

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

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

Figure 17: Can concurrency rules.

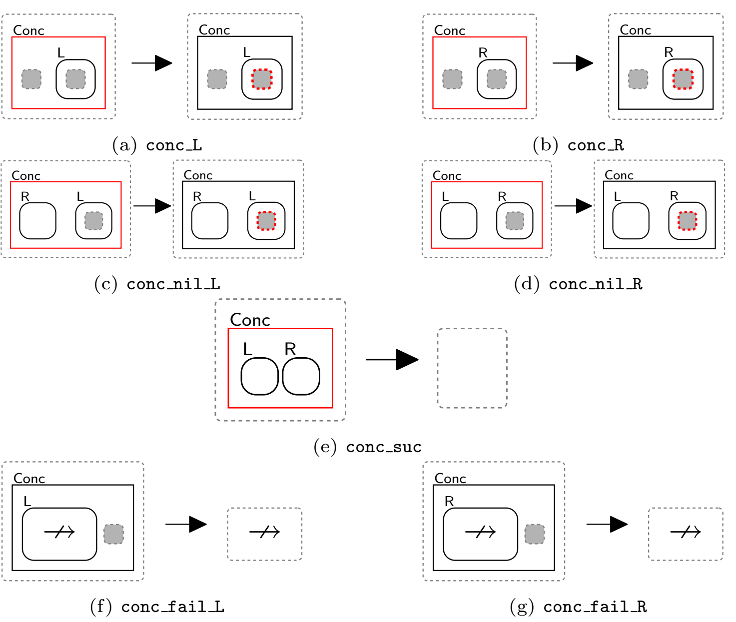


Figure 18: Reactions for concurrency with priorities: {conc_L, conc_R} < {conc_nil_L, conc_nil_R} < {conc_suc, conc_fail_L, con_fail_R}.

advantage of this approach is that all possible interleavings can be checked for correctness.

Two reaction rules `conc_L` (Fig. 18a), and `conc_R` (Fig. 18b) encode concurrency. As they have the same priority, these rules specify that reduction can be pushed down *either* the left or right branch.

Lemma 12. *(Faithfulness of $\|_1$ and $\|_2$) $\|_1$ and $\|_2$ have a corresponding finite reaction sequence $[\![\langle \mathcal{B}, P_1 \parallel P_2 \rangle]\!] \multimap\!\!\rhd^+ [\![\langle \mathcal{B}', P_1' \parallel P_2 \rangle]\!]$ and $[\![\langle \mathcal{B}, P_1 \parallel P_2 \rangle]\!] \multimap\!\!\rhd^+ [\![\langle \mathcal{B}', P_1 \parallel P_2' \rangle]\!]$, respectively.*

Proof. Consider $\|_1$, and assume $\langle \mathcal{B}, P_1 \rangle \to \langle \mathcal{B}', P_1' \rangle$, there are two cases:

Case 1. If $P_2 = nil$ then $\langle \mathcal{B}, P_2 \rangle \not\to$. The reaction rule `conc_nil_L` in Fig. 18c applies, resulting in $\mathsf{Reduce}.[\![P_1]\!]$, and by our assumption $[\![\langle \mathcal{B}, P_1 \rangle]\!]$ reduces.

Case 2. If $P_1 \neq nil$ then the reaction rule `conc_L` applies resulting in $\mathsf{Reduce}.[\![P_1]\!]$, and by our assumption $[\![\langle \mathcal{B}, P_1 \rangle]\!]$ reduces.

$\|_2$ is considered in a similar way. □

Concurrent programs are considered to have completed successfully when both branches complete, i.e. reduced to *nil*. The reaction rule `conc_suc`, given in Fig. 18e, handles the completion of concurrent programs.

Lemma 13. *(Faithfulness of $\|_\top$) $\|_\top$ has a corresponding finite reaction sequence $[\![\langle \mathcal{B}, nil \parallel nil \rangle]\!] \multimap\!\!\rhd^+ [\![\langle \mathcal{B}', nil \rangle]\!]$.*

Proof. $\multimap\!\!\rhd^+$ corresponds to $\xrightarrow{\texttt{conc_suc}}\!\!\rhd$. Trivial. □

In the case of failures, additional reaction rules `conc_fail_L` (Fig. 18f) and `conc_fail_R` (Fig. 18g) propagate failure up-the-tree if either of the two concurrent branches results in a failure. Importantly, we fail as soon one branch fails, rather than waiting for the other branch to complete (either successfully or with failure).

As before, a priority ordering on the reaction rules is required as some reaction rules generalise others, e.g. the reaction rule `conc_R` would also match reaction rule `conc_succ`.

6.2. Declarative Goals

Declarative goals allow an agent to respond *persistently* to event e until either the success or failure conditions are met. The CAN semantics for declarative goals are given in Fig. 19. The derivation rules G_s and G_f deal with the cases when either the success condition φ_s or the failure condition φ_f become true. The derivation 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. The derivation 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 φ_s nor φ_f becomes true).

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

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

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

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

Figure 19: Derivation rules for declarative goals.

To reduce the number of reaction rules for encoding declarative goals, we check both success and failure conditions simultaneously through reaction rule `goal_check` (Fig. 20a). As before, the entailment machinery provides atomic checks in both cases. Afterwards the reaction rules `goal_suc` (Fig. 20b) and `goal_fail` (Fig. 20c) determine if the goal should complete (either successfully or with failure). Strictly speaking it is possible both success/failure conditions hold simultaneously, however in practice it is usually assumed success/failure conditions are mutually exclusive.

An interesting feature of the Can derivation rule G_f is the use of ?*false* in the resulting state. This plays a similar role to ReduceF by explicitly creating an irreducible term to trigger further handling up-the-tree. Recall that the belief entailment in Can derivation rule ? can be regarded as the special case of the Can derivation rule *act* (Eq. (21)).

Therefore, we simply let `goal_fail` reduce to an Act with a *false* precondition (that always fails) to indicate a failure.

Lemma 14. *(Faithfulness of G_s) G_s has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f)\rangle]\!] \longrightarrow\!\!\rhd^+ [\![\langle \mathcal{B}, nil\rangle]\!]$.

Proof. $\longrightarrow\!\!\rhd^+$ corresponds to $\xrightarrow{\texttt{goal_check}}\!\!\rhd \xrightarrow{\{\texttt{set_ops}\}}\!\!\rhd^* \xrightarrow{\texttt{goal_suc}}\!\!\rhd$. The argument is similar to Lemma 1. □

Lemma 15. *(Faithfulness of G_f) G_f has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f)\rangle]\!] \longrightarrow\!\!\rhd^+ [\![\langle \mathcal{B}, ?false\rangle]\!]$.

Proof. $\longrightarrow\!\!\rhd^+$ corresponds to $\xrightarrow{\texttt{goal_check}}\!\!\rhd \xrightarrow{\{\texttt{set_ops}\}}\!\!\rhd^* \xrightarrow{\texttt{goal_fail}}\!\!\rhd$. The argument is similar to Lemma 1. □

Similar to the derivation rule $\rhd;$, the derivation rule $G_;$ reduces the left-branch of the symbol $\rhd$. The reaction `goal_reduce` (Fig. 20d) pushes the reduction down the left-branch. To ensure the ordering between rules G_s, G_f, and $G_;$, we explicitly match only on the case that the checks have already been performed. In other words, the goal will only be pursued if neither the success or failure condition holds.

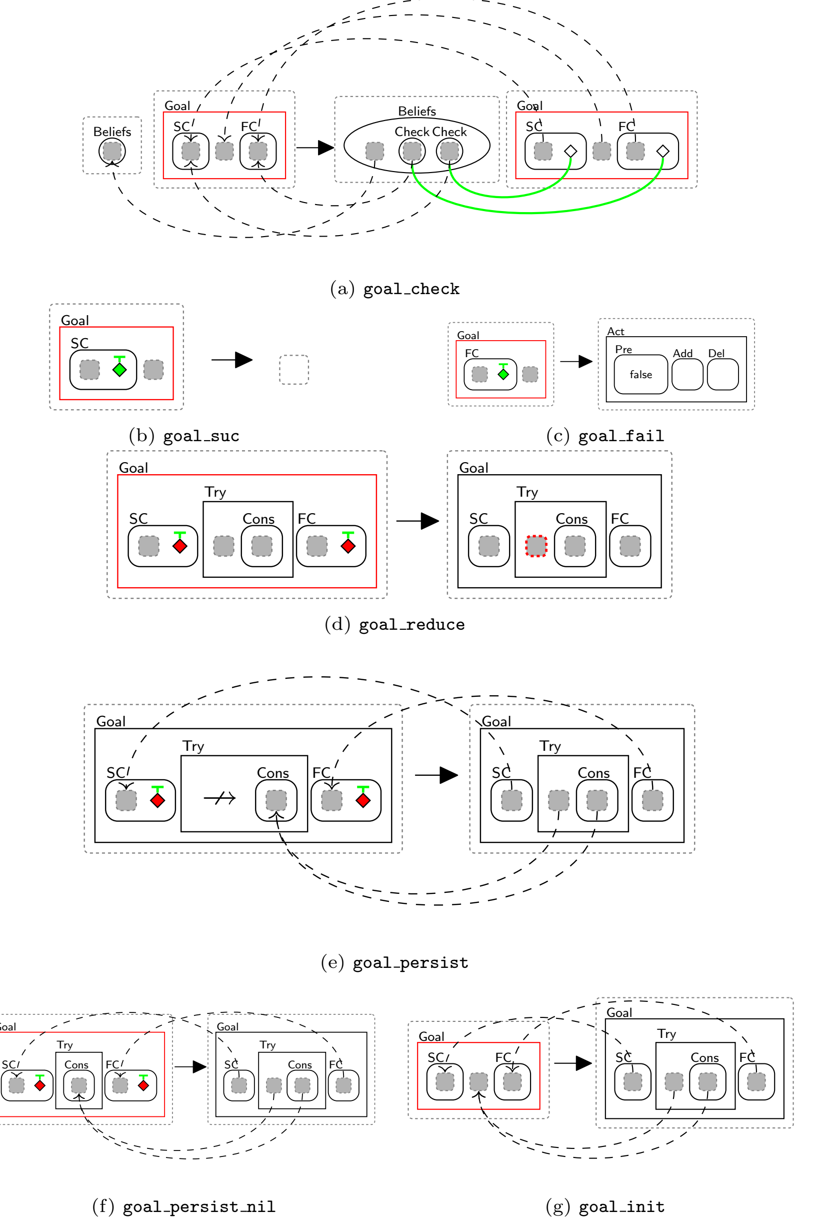


Figure 20: Reactions for declarative goals with priorities: `goal_init` $<$ {`goal_reduce`, `goal_check`, `goal_fail`, `goal_suc`} $<$ {`goal_persist`, `goal_persist_nil`}.

Lemma 16. *(Faithfulness of $G_;$) $G_;$ has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f)\rangle]\!] \multimap\!\!\rhd^+ [\![\langle \mathcal{B}', goal(\varphi_s, P_1' \rhd P_2, \varphi f)\rangle]\!]$.

Proof. Similar to Lemma 6. Assume $\langle \mathcal{B}, P_1\rangle \multimap\!\!\rhd^+ \langle \mathcal{B}', P_1'\rangle$. The initial state has form $[\![\mathcal{B}]\!] \parallel \mathsf{Reduce.Goal.}(\mathsf{SC.}[\![\varphi_s]\!] \mid \mathsf{FC.}[\![\varphi_f]\!] \mid \mathsf{Try.}([\![P_1]\!] \mid \mathsf{Cons.}[\![P_2]\!]))$ and so rule `goal_reduce` applies, which allows reduction of P_1 to P_1', with updated $\mathcal{B}'$.

□

The derivation rule $G_\rhd$ likewise is very similar to the derivation rule $\rhd_\bot$. Unlike in rule $\rhd_\bot$, however, in rule $G_\rhd$ we keep the $\rhd$ structure in-place and *replicate* P_2, thus giving the declarative goals their *persistence*. The Can reaction rule `goal_persist` (Fig. 20e) encodes this case with the match of ReduceF ensuring the premise $\langle \mathcal{B}, P_1\rangle \nrightarrow$ holds. Through duplication, we decouple the failure of the *plan* execution from the failure of the *goal* (as specified by success/failure conditions). Finally, an additional reaction `goal_persist_nil` (Fig. 20f) enables the agent to persist even in the case where the program executed successfully (but the goal success/failure did not hold).

Lemma 17. *(Faithfulness of $G_\rhd$) $G_\rhd$ has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, goal(\varphi_s, P_1 \rhd P_2, \varphi_f)\rangle]\!] \multimap\!\!\rhd^+ [\![\langle \mathcal{B}, goal(\varphi_s, P_2 \rhd P_2, \varphi_f)\rangle]\!]$.

Proof. Two cases.

Case 1. If $P_1 = nil$ then $\multimap\!\!\rhd^+$ corresponds to $\xrightarrow{\texttt{goal_persist_nil}}\!\!\rhd$. Trivial.

Case 2. If $\langle \mathcal{B}, P_1\rangle \nrightarrow$, then P_1 must eventually reduce to ReduceF. Rule $\xrightarrow{\texttt{goal_persist}}\!\!\rhd$ applies giving a result in the form $[\![\langle \mathcal{B}, goal(\varphi_s, P_2 \rhd P_2, \varphi_f)\rangle]\!]$ as required.

□

The derivation rule G_{init} is encoded through reaction `goal_init` that sets up the required $\rhd$ structure. To ensure this is applied at the right time, we have priority classes with $\texttt{goal_init} < \{\texttt{goal_persist}, \texttt{goal_reduce}\}$ to ensure the premise $P \neq P_1 \rhd P_2$ holds.

Lemma 18. *(Faithfulness of G_{init}) G_{init} has a corresponding finite reaction sequence* $[\![\langle \mathcal{B}, goal(\varphi_s, P, \varphi_f)\rangle]\!] \multimap\!\!\rhd^+ [\![\langle \mathcal{B}, goal(\varphi_s, P \rhd P, \varphi_f)\rangle]\!]$.

Proof. $\multimap\!\!\rhd^+$ corresponds to $\xrightarrow{\texttt{goal_init}}\!\!\rhd$. Priority classes of reactions ensure $P \neq P_1 \rhd P_2$, as required. □

For declarative goals, due to persistence, there is no rule that propagates failures upwards[7].

As with the previous lemmas, these extended features can be integrated easily into Theorem 1 to prove the extended semantics is also faithful.

[7]Meeting the failure conditions does eventually lead to failure but this requires additional steps.

Theorem 2 (Faithfulness (extended))**.** *For each* CAN *step (including features of concurrency and declarative goal)* $\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E'^e, \mathcal{B}', \Gamma'\rangle$ *there exists a corresponding finite sequence of reactions such that*

$$[\![\langle E^e, \mathcal{B}, \Gamma\rangle]\!] \longrightarrow^{+} [\![\langle E'^e, \mathcal{B}', \Gamma'\rangle]\!] \ .$$

Proof. Follows from Lemmas 1–18. □

7. UAVs Examples

To illustrate our modelling and verification framework, we consider three examples taken from UAV surveillance and retrieval mission systems. The examples cover persistent patrol, concurrent sensing, and contingency handling in object retrieval and highlight the three distinguishing features of CAN: declarative goals, concurrency, and failure recovery.

7.1. Persistent Patrol

BDI Agent Design for Persistent Patrol

```
1  // Initial beliefs
2  ¬battery_low, ¬harsh_weather
3  // External events
4  e_init1
5  // Plan library
6  e_init1 :  true <- goal(false, e_patrol_task, false)
7  e_patrol_task :  true <- goal(sc, e_patrol, false); e_pause
8  e_patrol :  true <- patrol
9  e_pause :  battery_low <- request; wait; charge
10 e_pause :  harsh_weather<- activate_parking
```

where sc = harsh_weather ∨ battery_low.

Bigraph Encoding

big persistent_patrol =
Beliefs.(B(1) | B(2)) ‖ Desires.E_{e1} ‖ Intentions.1
‖ Plans.(
PlanSet$_{e_init1}$.(Plan.(Pre.1 | PB.Goal.(SC.False | E_{e_task1} | FC.False)))
| PlanSet$_{e_patrol_task}$.(Plan.(Pre.1 | PB.(Seq.(Goal.(SC.B(3) | E_{e_patrol} | FC.False) | Cons.E_{e_pause}))))
| PlanSet$_{e_patrol}$.(Plan.(Pre.1 | PB.$[\![patrol]\!]$))
| PlanSet$_{e_pause}$.(
| Plan.(Pre.B(4) | PB.(Seq.($[\![request]\!]$ | Cons.(Seq.($[\![wait]\!]$ | Cons.$[\![charge]\!]$)))))
| Plan.(Pre.B(5) | PB.$[\![activate_parking]\!]$)))
where B(1) = ¬battery_low, B(2) = ¬harsh_weather, B(3) = sc,
B(4) = battery_low, and B(5) = harsh_weather.

Figure 21: Persistent patrol: BDI agent design and bigraph encoding.

UAVs are used in surveillance operations, with a UAV patrolling a predefined area to identify objects of interest. The UAV can request refuelling

BDI Agent Design for Concurrent Sensing in One Intention

```
1  // Initial beliefs
2  ram_free, storage_free
3  // External events
4  e_init2
5  // Plan library
6  e_init2 :  true <- e_dust|| e_photo
7  e_dust :  ram_free ∧ storage_free <- collect_dust; analyse; send_back
8  e_photo :  ram_free ∧ storage_free <- focus_camera; save_shots; zip_shots
```

Bigraph Encoding

big concurrent_sensing =

Beliefs.(B(6) | B(7)) ‖ Desires.E_{e_init2} ‖ Intentions.1

‖ Plans.(

PlanSet$_{e_init2}$.Plan.(Pre.1 | PB.(Conc.(L.E_{e_dust} | R.E_{e_photo})))

| PlanSet$_{e_dust}$.Plan.(Pre.(B(6) | B(7)) | PB.(Seq.($[\![collect_dust]\!]$ | Cons.(Seq.($[\![analyse]\!]$ | Cons.$[\![send_back]\!]$)))))

| PlanSet$_{e_photo}$.Plan.(Pre.(B(6) | B7) | PB.(Seq.($[\![focus_camera]\!]$ | Cons.(Seq.($[\![save_shots]\!]$ | Cons.$[\![zip_shots]\!]$))))))

where B(6) = `ram_free` and B(7) = `storage_free`.

Figure 22: Concurrent sensing in one intention: BDI agent design and bigraph encoding.

when the battery is low, and parking mode should be activated when there is harsh weather.

The agent design and its corresponding bigraph encoding is in Fig. 21. The external event `e_init1` (line 4) initiates persistent patrol. There is only one plan (line 6) relevant to `e_init1`, whose context is always true (represented by an empty region bigraph 1), thus always applicable, and whose plan-body is declarative goal `goal(false,e_patrol_task,false)`. The event `e_patrol_task` is persistent because the success and failure conditions never hold, that is, we have an infinite process executing `e_patrol_task`. In practice, we require some flexibility in case of low battery or harsh weather. The plan for `e_patrol_task` (line 7) indicates the patrol task may need to be paused (i.e. followed by the event `e_pause`), when the success condition is true, i.e. when `battery_low` or `harsh_weather` holds (added to the belief base). If the pause is required and after achieving event `e_pause` (lines 9-10), the event `e_patrol_task` will be pursued again. For succinct presentation, we note that the encoding of action such as `patrol` and `wait` are not shown, but can be found in our model [30].

7.2. Concurrent Sensing in One Intention

UAVs may also be used for sensing tasks. In this case we consider a UAV that analyses dust particles, and performs aerial photo collection, e.g. for analysis in post volcanic eruptions.

An agent design to achieve this concurrent sensing task is in Fig. 22. The external event `e_init2` (line 4) initiates the mission and the relevant plan (line 6) has tasks for dust monitoring (`e_dust`) and photo collection (`e_photo`) as the concurrent programs in the plan-body. The on-board dust sensors require

BDI Agent Design for Concurrent Sensing with Two Intentions

```
1  // Initial beliefs
2  ram_free, storage_free
3  // External events
4  e_dust, e_photo
5  // Plan library
6  e_dust :  ram_free ∧ storage_free <- collect_dust; analyse; send_back
7  e_photo :  ram_free ∧ storage_free <- focus_camera; save_shots; zip_shots
```

Bigraph Encoding

big concurrent_sensing =

Beliefs.(B(6) | B(7)) ‖ Desires.(E_{e_dust} | E_{e_photo}) ‖ Intentions.1

‖ Plans.(

$\mathsf{PlanSet}_{e_dust}$.Plan.(Pre.(B(6) | B(7)) | PB.(Seq.($[\![collect_dust]\!]$ | Cons.(Seq.($[\![analyse]\!]$ | Cons.$[\![send_back]\!]$)))))

| $\mathsf{PlanSet}_{e_photo}$.Plan.(Pre.(B(6) | B7) | PB.(Seq.($[\![focus_camera]\!]$ | Cons.(Seq.($[\![save_shots]\!]$ | Cons.$[\![zip_shots]\!]$))))))

where B(6) = `ram_free` and B(7) = `storage_free`.

Figure 23: Concurrent sensing in two intentions: BDI agent design and bigraph encoding.

high-speed RAM to collect and analyse the data, hence condition `ram_free`, and when the analysis is complete, results are written to storage (hence condition `storage_free`), and sent back to the control. Similarly, to collect aerial photos, the UAV reserves and focuses the camera array (`focus_camera`), then camera shots are compressed (`zip_shot`), and sent back. Recall, for successful completion, both concurrent tasks have to complete successfully.

7.3. Concurrent Sensing in Two Intentions

In Can, an agent can execute multiple intentions concurrently in an interleaved manner. As an example of concurrency between intentions, we revise the task of concurrent sensing to use two different intentions. The design of this scenario is given in Fig. 23. We see that, compared to the agent design in Fig. 22, we now use two separate external events—`e_dust` and `e_photo`—resulting in two intentions. We reflect on the inability of verifying multiple intentions in Can, and detail the difference of concurrency *within* an intention and concurrency among multiple intentions in Section 8.

7.4. Contingency Handling for a Retrieve Task

UAVs may be used for object retrieval tasks, e.g. package delivery. An agent design for retrieval is in Fig. 24. It has one (retrieve) task, initiated by external event `e_retrv` (line 4), which may be affected by engine or sensor malfunction, Event `e_retrv` is handled by five relevant plans available (lines 6 to 10). The first 3 plans provide different flight paths after take-off, in which case the failure condition is (subsequent) engine or sensor malfunction. The last 2 plans (line 9 and 10) indicate safe recovery in the event of engine or sensor malfunction.

BDI Agent Design of Retrieve Task

```
1  // Initial beliefs
2  ¬sensor_malfunc, ¬engine_malfunc
3  // External events
4  e_retrv
5  // Plan library
6  e_retrv :  φ <- take_off; goal(at_destination, e_path1, fc); retrieve
7  e_retrv :  φ <- take_off; goal(at_destination, e_path2, fc); retrieve
8  e_retrv :  φ <- take_off; goal(at_destination, e_path3, fc); retrieve
9  e_retrv :  sensor_malfunc <- return_base
10 e_retrv :  engine_malfunc <- activate_parking; send_GPS
11 e_path1 :  true <- navigate_path_1
12 e_path2 :  true <- navigate_path_2
13 e_path3 :  true <- navigate_path_3
```

where $\varphi = \neg$`sensor_malfunc` $\wedge \neg$`engine_malfunc`, `fc` $=$ `sensor_malfunc` $\vee$ `engine_malfunc`

Bigraph Encoding

big retrieve_task =

Beliefs.(B(8) | B(9)) ‖ Desires.E_{e_retrv} ‖ Intentions.1

‖ Plans.(

PlanSet$_{e_retrv}$.(

Plan.(Pre.(B(8) | B(9)) | PB.(Seq.(⟦*take_off*⟧ | Cons.(Seq.(Goal.(SC.B(10) | E_{e_path1} | FC.B11) | Cons.⟦*retrieve*⟧)))))

| Plan.(Pre.(B(8) | B(9)) | PB.(Seq.(⟦*take_off*⟧ | Cons.(Seq.(Goal.(SC.B(10)) | E_{e_path2} | FC.B(11) | Cons.⟦*retrieve*⟧)))))

| Plan.(Pre.(B(8) | B(9)) | PB.(Seq.(⟦*take_off*⟧ | Cons.(Seq.(Goal.(SC.B(10) | E_{e_path3} | FC.B(11)) | Cons.⟦*retrieve*⟧)))))

| Plan.(Pre.B(12)) | PB.⟦*return_base*⟧)

| Plan.(Pre.B(13)) | PB.(Seq.(⟦*activate_parking*⟧ | Cons.⟦*send_GPS*⟧))))

| PlanSet$_{e_path1}$.Plan.(Pre.1 | PB.(⟦*navigate_path_1*⟧))

| PlanSet$_{e_path2}$.Plan.(Pre.1 | PB.(⟦*navigate_path_2*⟧))

| PlanSet$_{e_path3}$.Plan.(Pre.1 | PB.(⟦*navigate_path_3*⟧)))

where B(8) = ¬`sensor_malfunc`, B(9) = ¬`engine_malfunc`, B(10) = `at_destination`, B(11) = `fc`,
B(12) = `sensor_malfunc`, and B(13) = `engine_malfunc`.

Figure 24: Retrieval contingency: BDI agent design and bigraph encoding.

7.5. Properties

To verify the designs, we generate a transition system from the BRS representing the agents (and their semantics). The transition system has bigraphs as states and reactions as transitions. We can reason about static properties using *bigraph patterns* [22] and dynamic properties using linear or branching time temporal logics such as Computation Tree logic (CTL) [33], which we use in our examples. As we simply generate a transition system, the property specification language is ultimately constrained by the logics the selected model checker supports. For example, in our case we can use the non-probabilistic and non-reward logics provided in PRISM.

7.5.1. Bigraph patterns

Bigraph patterns are predicates on states: if the pattern *matches* the current state then the predicate is true.

We have found the bigraph patterns most useful for reasoning about BDI agents are often a fragment of the right-hand side of reactions, i.e. they check that a desired or anticipated operation has taken place. For example, consider the state predicate: there is a declarative goal corresponding to event `e_patrol_task` (i.e. `goal(false, e_patrol_task, false)`). The bigraph pattern is

$$\mathsf{Goal}.(\mathsf{SC}.(\mathsf{False} \mid \mathsf{id}) \mid \mathsf{FC}.(\mathsf{False} \mid \mathsf{id}) \mid \mathsf{Try}.\mathsf{id})$$

where SC is the success condition, FC the failure condition, and Try the plan choice $\triangleright$. The presence of Try indicates that event `e_patrol_task` is within the given declarative goal and has been reduced to its set of relevant plans, from which an applicable plan is selected, according to the right-hand side of the reaction given in Fig. 12b. As long as Try is present (regardless of what is under it, i.e. $\mathsf{Try}.\mathsf{id}$), the declarative goal is being pursued.

7.5.2. Example properties

Example 1 (Persistent patrol). A key property is that the goal corresponding to event `e_patrol_task` is persistent: $\mathbf{A}[\mathbf{G}\ \mathbf{F}\varphi_1]$, where

$$\varphi_1 \stackrel{\text{def}}{=} \mathsf{Goal}.(\mathsf{SC}.(\mathsf{False} \mid \mathsf{id}) \mid \mathsf{FC}.(\mathsf{False} \mid \mathsf{id}) \mid \mathsf{Try}.\mathsf{id})$$

As expected the property holds.

Example 2 (Concurrent sensing in one intention). A useful property to investigate is whether it is possible to complete both sensing tasks regardless of their interleaving. Recall that in CAN semantics, whenever an intention is completed or fails, the agent will simply remove it from the intention base (A_{update} Fig. 15). Therefore, to make sure that an intention is successfully achieved, we have to ensure that a given intention is indeed removed only *after* being completed successfully. We denote the bigraph pattern for the successful completion of a given intention as $\varphi_2 \stackrel{\text{def}}{=} \mathsf{Intent}.1$, the failure of completion of an intention $\varphi_3 \stackrel{\text{def}}{=} \mathsf{Intent}.\mathsf{ReduceF}$, and the removal of an intention from intention base

$\varphi_4 \stackrel{\text{def}}{=}$ Intentions.1. Bigraph pattern φ_4 specifies the removal of an intention from the intention base if and only if it is the only intention in the base. If there is more than one intention, it is impossible to reason about which intention is removed because there are no intention identifiers in CAN. We reflect on this lack in Section 8.6. Given the bigraph patterns above, the property is $\mathbf{A}[\mathbf{F}(\varphi_2 \wedge \mathbf{X}\varphi_4)]$. This property is false, and we find that $\mathbf{E}[\mathbf{F}(\varphi_3 \wedge \mathbf{X}\varphi_4)]$ holds (i.e. there exists a path for which eventually the intention is removed after being failed). This is because concurrency can introduce undesirable race conditions[8]. For example, the action `send_back` in line 7 needs to be executed before the action `save_shots` is executed, to free required storage. This example highlights the benefits of a formal model for analysis at design time.

Example 3 (Contingency handling). Similar to Example 2, a desirable property is that regardless of any malfunction, the intention for event `e_retr` is removed after successful completion. The property is $\mathbf{A}[\mathbf{F}(\varphi_2 \wedge \mathbf{X}\varphi_4)]$ and it holds.

Before we give the results of verification, recall that our bigraph encoding introduces intermediate states that do not correspond to an agent step. Therefore, the operator $\mathbf{X}$ (*next*) has to be used carefully; some properties may require modification, because e.g. the *next* operator refers to the next internal state, not the next agent state. For example, there may be belief checks between agent steps. In examples 2 and 3 above, no modifications were required.

7.6. Results

For automatic verification we exported the transition system to the PRISM model-checker[9] by assuming all transitions occur with equal probability. The size of transition system[10] generated by BigraphER and verification times are as follows:

Example	**States**	**Transitions**	**Build time (s)**	**Ver. time (s)**
Persistent patrol	239	287	7.30	0.081
Concurrent sensing (1 Intent)	731	879	20.47	0.01
Concurrent sensing (2 Intents)	856	1074	15.11	N/A
Contingency handling	644	922	79.98	0.002

While contingency handling has fewer states/transitions than the concurrent sensing example, it takes more time to generate the transition system. We attribute this to the former containing more bigraph entities. Similarly, while concurrent sensing in two intentions has more states/transition than its one intention counterpart, it takes less time to generate the transition system due to having fewer bigraph entities.

[8]This race condition is within the agent design itself, and should not be confused with the race between reaction rules that update the belief sets (a bigraph model implementation detail).

[9]Currently the only model-checker format supported by BigraphER.

[10]Build times were obtained on a laptop with a 16-core Intel Core i7-11800H at 2.30GHz (hyperthreaded), 16 GB memory, and running 64-bit Ubuntu Linux 20.04.3 LTS.

8. Reflections

We reflect on the insights gained into the CAN language through the process of building the bigraph model and detail our first-hand experience of the theoretical and practical value of Bigraphs for encoding agent languages. We stress that our reflections on CAN should not be taken as criticism of CAN in any sense. On the contrary, we hope to show that the explicitness of the bigraph encoding is useful, e.g. to show areas of semantics with too much (resp. too little) information, and to aid in the continuous advancement of BDI family languages, in particular, from the point of verification and validation.

8.1. Modularity in Semantics of CAN

A distinguished characteristic of CAN (similar to Modular Structural Operational Semantics (MSOS) [34]) is that the transition rules for each constructcan be given incrementally, i.e. a modular operational semantics. In this case, the modularity in CAN (same as in 3APL) separates how to evolve an intention (i.e. the intention-level semantics) from how to evolve the whole agent (i.e. the agent-level semantics). This approach has its merits, for example, we can easily extend or modify one side of the semantics (e.g. the agent-level) without altering the other one. This was illustrated when adding the concurrency and declarative goals extensions (Section 6). The extensions only change intention-level steps, and as such, do not affect the overall faithfulness theorem as this is defined over agent-level steps.

The two-levels of semantics could be useful for verification. For example, we may consider *only* the agent-level transitions which would give snapshots of the agent state, without any information on *how* choices were made. In the bigraph model we do not make a distinction between agent-level and intention-level transitions, and both appear in the resulting output. The bridging of agent-level and intention-level transitions is performed by the introduction of the Reduce entity. For example, the reaction rule `intention_step` that encodes the agent-level derivation rule A_{steps} introduces a Reduce entity to an intention, requesting it to be reduced according to the intention-level semantics. The use of instantaneous reaction rules, that do not show up in the resulting transition system, would allow only agent-level steps to be analysed without changes to the reaction rules themselves.

8.2. Inconsistency of Semantics in CAN *Literature*

In the literature, there are subtle differences between definitions of the CAN semantics. In particular, between [6] and [23]. For example, consider the $\triangleright_\bot$ rule from the two works above:

$$\frac{P_1 \neq nil \quad \langle \mathcal{B}, P_1 \rangle \nrightarrow}{\langle \mathcal{B}, P_1 \triangleright P_2 \rangle \rightarrow \langle \mathcal{B}, P_2 \rangle} \quad \triangleright_\bot \text{ in [6]}$$

$$\frac{P_1 \neq nil \quad \langle \mathcal{B}, P_1 \rangle \nrightarrow \quad \langle \mathcal{B}, P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle}{\langle \mathcal{B}, P_1 \triangleright P_2 \rangle \rightarrow \langle \mathcal{B}', P_2' \rangle} \quad \triangleright_\bot \text{ in [23]}$$

In [6], the rule $\rhd_{\perp}$ is only dependent upon the irreducibility of the program P_1. However, in [23], not only is it dependent upon the irreducibility of P_1, but, within the same operation, the reducibility of P_2.

This change is significant as, in the first case, we wait to do failure recovery. This can allow the current belief base to be updated before selecting a new plan (in all cases P_2 has the form $e : (|\Delta|)$). In the second case there is no scope to wait for belief base changes.

It is not immediately clear which approach is better in practice. One benefit of a formal model is that we can begin to unpick these questions by substituting the current `try_failure` reaction for a modified version.

8.3. Redundant Event Names

The CAN language includes the form $e : (|\Delta|)$ representing a set of relevant plans which can be used to address the event e. This set is updated as plans are selected and executed. For example, when an applicable plan is selected (i.e. $\varphi : P \in \Delta$ and $\mathcal{B} \models \varphi$), it will be removed from the set of remaining plans (i.e. $e : (|\ \Delta \setminus \{\varphi : P\}\ |)$). However, after a set of relevant plans is selected from the plan library, the event name e becomes *redundant* in the sense that it is never used by any CAN semantic rules. This is seen clearly in the bigraph model, where only `reduce_event` (in Fig. 11) utilises the event name link. Other rules always match the event name as open (connected to 0 or more other entities). This suggests that the form of plans within the plan library, and those within intentions should be different, e.g. $e : (|\Delta|)$ and $(|\Delta|)$.

8.4. No Difference between Intention Success and Failure

As a high-level planning language, CAN remains agnostic to many important issues in practice. One such issue is the inability to tell if an intention completed successfully, or with a failure. The derivation rule A_{update} in Fig. 6 simply removes a completed intention from the intention base, namely an intention *nil* or one that is failed and cannot make any further transition. Therefore, the completion of an intention is not equivalent to the achievement of an intention. To verify the achievement of an intention (which in practice is the most important property to check), we also have to ensure that its completion is not due to the failure. This is precisely how we verify the achievement of an intention in Section 7.5.2. Therefore, in our bigraph model, we have to encode the derivation rule A_{update} into two cases, namely `intention_done_F` for failure case and `intention_done_succ` for success case.

8.5. Oracle for Failure

Failure in CAN semantics is denoted by $\langle \mathcal{B}, P_1 \rangle \nrightarrow$ as the negative premise in the related derivation rule (e.g. $\rhd_{\perp}$). Therefore, to be able to apply the rule $\rhd_{\perp}$, the agent somehow can "look-ahead" to the result of the inner-reduction, i.e. there is some oracle that determines if the inner-reduction is possible. However, in practice (e.g. our bigraph encoding), no oracle exists, and the agent has to explicitly to try progress a step to see if it reduces. In others words, unlike the

derivation rules which, to some extent, have the impression the failure occurs via one single rule $\langle \mathcal{B}, P_1 \rangle \nrightarrow$, it actually involves a strict partial execution of other rules. It also explains why we convert the negative premise into the positive premise in the actual encoding with additional token `ReduceF`. It can be clearly seen in Fig. 16 where, before an action is deemed as un-executable, its pre-condition has to be actually checked to be false according to the belief base.

8.6. Absence of Meta-level Reasoning

While it is possible to reason about agents when only a single intention is involved – for example through checking a property that checks if an intention failed before it was removed (see Section 7.5.2) – these approaches do not apply when there are more than one concurrent intentions. The main issue here is that intentions lack identifiers. If we want to stay within the semantics in CAN, approaches to identifying specific intentions include (1) fixing the last program within any intention to be unique to allow checking when this specific program is removed, or (2) ensuring actions add unique beliefs however this requires knowing ahead of time the actions that will be executed in success/failure cases.

Ideally, intentions would have unique identifier to aid verification. Adding such an identifier is straightforward by replacing $P \in \Gamma$ with $\langle identifier, P \rangle \in \Gamma$. As such, to track an intention is removed, we simply have a bigraph pattern $\mathsf{Intent.}(\mathsf{identifier} \mid \mathsf{id})$ where $\mathsf{identifier}$ is the identifier of the intention and id the site that abstracts away specific details of the intention. We argue that by indexing the intentions and further labelling its status (e.g. active or suspended)—tackled by some promising work [35, 36]—that allows reasoning on intentions provides strong starting point for next level of agent verification, e.g. in the context of interacting with users.

Another area where keeping meta-information available is useful is to allow tracking events to the intentions that are handling those events. In the current semantics, when an event is processed (by A_{event}) it is removed completely from the desires structure and replaced by the set of relevant plans within an intention. As we execute the plans we lose track of which event e generated that intention (i.e. the means-end relations).

8.7. Concurrency Within vs. Among Intentions

One important decision and agent designer needs to make is whether to use internal concurrency (through $\|$) or intention concurrency via multiple events. Without appropriate intention labelling (discussed in Section 8.6), it remains difficult to write formal verification properties when using multiple-intentions. In practice, concurrency among intentions requires significantly more state transitions than its counterpart seen in Section 7.6. This should not come as a surprise as in concurrency among two intentions, the agent has two choices (as to decide which intention to progress) on each step, while the agent only makes a decision to progress the left or right part of concurrency within an intention. That is, there is more interleaving to analyse with the multiple intentions.

8.8. Experience on Theoretical and Practical Bigraph Encoding Approach

Finally, we reflect on bigraphs as an (agent) language encoding framework from both a theoretical and a practical perspective.

We found bigraphs to be useful and easy to use for encoding the syntax of CAN. It required a modest number of (core) entities (Table 3) and there was a very direct translation of CAN syntax to bigraphs (Fig. 3) thanks in part to the inductively defined (compositional) nature of bigraphs. One benefit, often not seen in other modelling formalisms, is the use of parallel regions to separate models into different, but interacting, perspectives. We had four perspectives: *Belief*, *Desire*, *Intention* and *Plan* and this helped to separate concerns and make the encoding process easier to manage. We expect perspectives to play a large role in extending the model, for example, adding an *Environment* perspective to model when external events can happen. The use of links in bigraphs has been proven a useful feature, allowing and event to directly connect to the set of plans that can respond to it. Using links decreases the likelihood of human errors, e.g. misspelling of event names

For encoding semantics, allowing user-specified reaction rules facilitated a direct mapping of transition rules of CAN, which led to the establishment of a faithful encoding where it was possible to sketch proofs based on finite rule sets. One area where we found bigraphs particularly useful was analysing the treatment of concurrency within and among multiple intentions (Section 8.7). Being able to draw the rules diagrammatically proved highly useful for explaining the model to others, and noticing potential errors at a glance (much more than text based syntax, although this is anecdotal evidence only at this point).

For debugging, the graphical output of each state in the transition system provided by BigraphER provided a highly visual debugging experience and enabled us to locate the bugs with ease. However, we found that using reaction rule priorities can often make it difficult to know exactly what rules can fire when, and this can lead to subtle bugs. Another disadvantage is that, as bigraphs are general purpose, we often have to provide extra rules for operations that are built-in for other formalisms, e.g. set operations. In terms of performance, BigraphER performs well in general, but, due to matching semantics, the time required to generate a transition system is dependent on both the number of transitions and the number of bigraph entities in the agent plan library (potentially large for complex agents). However, our latest work [37] which utilises a subgraph isomorphism solver improved the matching performance by over two orders of magnitude on a range of problem instances drawn from real-world mixed-reality, protocol, and conference models.

For verification, unlike other approaches, e.g. both Maude-based and AIL-based approaches that rely on their own dedicated model checkers, we instead export a predicate-labelled transition system for use with existing model checking tools such as PRISM. This allows a wide range of highly-performance tools, including different logics etc., to be used, however it does limit, for example, the amount of symbolic analysis these tools can perform, leading to degraded performance.

Overall we would recommend bigraphs as a tool for working with programming languages. We believe there is much to be gained including from the diagrammatic notation, explicit entity linking, multi-perspective modelling, and efficient tooling.

9. Related Work

Reasoning about BDI agents through model checking has been well explored. A key work in this area [12] reports a translation of AgentSpeak programs to both the Promela modelling language and Java, and shows how to apply the Spin [14] model checker and Java PathFinder program model checker to verify the agents. Similar to our bigraph encoding, the translation of AgentSpeak programs in Promela in [12] provides an encoding the semantics of AgentSpeak(F)—the finite state version of AgentSpeak—in Promela. However, there is no formal faithfulness establishment of such a translation provided and it is limited to the AgentSpeak language that does not contain features such as declarative goals. The translation of AgentSpeak programs to Java facilitates direct verification of the implementation of an agent rather than an abstract model specification (as with Promela) by symbolically analysing the underlying Java bytecodes. In both cases, the proprieties checked are specified in a simplified BDI logic language mapped down in linear temporal logic (LTL) formula [38].

Comparing the translation experience of these two translation in [12], Java stood out as a more promising approach (as a general purpose language) compared to Promela (often used for the verification of communication protocols). Many have built upon this Java-based verification approach. In particular, recent work implements a BDI agent programming infrastructure as a set of Java classes – the Agent Infrastructure Layer (AIL) [15]. As a matter of fact, the Gwendolen BDI language [39] provides the default semantics for the AIL, and is designed with verification in mind by including extra book-keeping and transition rules that purely assist verification.

The AIL has been further developed [40] to support the verification of *heterogeneous* multi-agent systems by allowing different agent programming languages to be used within the same AIL framework. Although the AIL supports heterogeneous agents, to date the BDI programming languages implemented in the AIL [41] is tightly bounded to Gwendolen and its extensions (e.g. [42]) along with another language named GOAL [43]. Crucially, what these approaches verify is the *implementation* of a given language. The faithfulness of the implementation to the language semantics is often omitted for convenience. Utilising Java PathFinder (and its enhanced version [40]) has the advantage of bypassing the need of a mathematical model by deriving the model directly from the program codes. However it typically suffers from a significant performance bottleneck due to the symbolic execution of Java bytecode. Agent properties for AIL are usually specified in LTL fashion, There is, however, an exception in [44] where the model generated by Java PathFinder is converted to the input

language of PRISM [45] to, e.g. provide access to probabilistic property specification. Unfortunately the conversion to PRISM does not maintain direct link between the implemented program and the model being verified, e.g. it might be difficult to reflect back into the application when creating counter examples. Meanwhile, by simplifying the structure and execution of AgentSpeak (deviating from mainstream BDI agents), it can also facilitate the verification of probabilistic and time bound properties through PRISM [46]. Finally, there is promising progress to verify the hybrid autonomous system in which the high-level is discrete logic-based framework (modelled by BDI agents) and the low-level is a continuous control system [47].

The two main BDI languages implemented in AIL are Gwendolen and GOAL. Unlike main-stream BDI programming languages, e.g. AgentSpeak, GOAL is a pure reactive system and does not select pre-defined plans from a library but instead selects individual actions (or a sequence of actions). Like CAN, Gwendolen handles declarative goals, failure recovery and concurrency with some differences. In Gwendolen, declarative goals make statements about the beliefs the agent wishes to hold and remains a goal until the agent gains the appropriate beliefs. As such, the declarative information in Gwendolen is only carried for the initial goal of the intention, no declarative information is carried for any of its active sub-events. For example, if beliefs sought hold, the sub-event will still be executed to the end. Meanwhile, in CAN, the declarative information is carried for any stage of evolution of programs in the declarative goal. Once the success condition holds, the related program is halted immediately. Gwendolen does not allow goal failure conditions so is unable to decouple goal failure from plan failure. For failure recovery, Gwendolen is explicitly programmed with the appropriate plan revision rules (as meta-level rules) which specify a prefix of the current plan to be dropped and replaced by another. Finally, concurrency in Gwendolen is only allowed in the intention level (i.e. no concurrent execution within an intention), and, by default, is conducted in first-in-first-out fashion to manage interleaving.

Work exists where the operational semantics of (general) agent programming languages is explicitly encoded directly in verification languages. For example, [48] presents a programming language for multi-agent systems, MABLE, that is translated to Promela and verified using Spin. Another work [49] develops a verification framework for multi-agent systems specified by the cognitive agents specification language (CASL). This framework is based on the prototype verification system (PVS) and facilitates theorem to verify properties of CASL specifications. Both these agent languages are not specifically geared towards BDI-style rational agents but provide more general tools for the analysis of agents. None of these works provides a formal establishment of the faithful translation between the given agent language and verification language.

Besides Promela, term-rewriting, specifically in Maude [21], has been used to encode BDI agent languages, allowing verification of temporal properties with the Maude LTL model checker [50]. For example, Maude has been used to directly encode GOAL semantics in a single agent setting [51]. In a multi-agent setting, [52] shows how both Jason and 3APL programs can be translated into

the language meta-APL and then encodes meta-APL semantics in Maude for subsequent verification. Besides the approaches based on existing verification tools/model checkers (including Spin, Java PathFinder, and Maude), there is a different verification approach [53] that performs the verification by modelling the interpreter (i.e. the implementation) of the GOAL language. The agent interpreter is used to generate the state space and a model checker is built on top of this interpreter with two components: 1) a translation the linear temporal properties to a property space, and 2) a means to evaluate the property using a search over both the property and state spaces. The authors provide an empirical comparison for the GOAL language of its interpreter-based, AIL-based, and Maude-based verification approach. Interestingly, it found that the Maude-based verification approach was unable to deal even with the simple toy examples due to high verification times.

Recent work continues to advance the state-of-the-art of agent verification problems. For example, there has been considerable work on developing various state-space reduction techniques to improve the efficiency of verification and support richer property specifications for large agent systems. The work [54] applies program slicing techniques—that have been successfully used in conventional programming languages—to reduce the state-space required in agent program verification problem. The slicing technique eliminates details of the program that are not relevant to the property being analysed, i.e. property-based slicing. This work was extended [55] to provide detailed correctness and complexity results for a property-based slicing algorithm for AgentSpeak. The work [56] takes this even further by proposing a new, and improved, slicing method. Noticeably, there is another work [57] that combines two state-space reduction techniques: property-based slicing and partial order reduction for verifying the GOAL language.

Bigraphs have been shown [24] to be suitable for encoding process algebras such as CCS [58], Mobile Process [59], and π-calculus [18] as well as the Actor programming model [60]. Recently, there is also a growing trend to specify and verify agent-based systems via bigraphs, in particular, multi-agent systems. However, most of them still remain at the stage of proof of concept. For example, the work [61] proposes a methodology for modelling and simulating multi-agent systems via bigraphs. The core idea is that the containment relation of bigraphs mirrors the administrative relations of agents while reaction rules model agent reconfigurations, e.g. bigraph destruction translates into agent termination. One work that is perhaps closest to ours is [62], which also models BDI agents via bigraphs. However, it considers multi-agent systems, and treats the internal reasoning of each BDI agent as a black box. As a result, they provide no details regarding how the agents behave in an environment.

10. Future Work

The encoding of BDI agents in bigraphs is our first step laying out a foundation for more advanced reasoning. As future work, we have in mind two main

extensions: probabilistic reasoning, in particular, plan selection and intention trade-off, and dynamic environments.

In general, there may be several applicable plans which achieve a given event. The agent has to select one and it may be desirable to specify what is "most appropriate" at that time, which may depend upon different, and possibly domain-specific characteristics, e.g. cost and preference. Additionally, the agent may be pursuing a set of concurrent intentions, i.e. there is concurrency between the top-level external events. Similar to the plans, intention, it may be desirable to again specify "most appropriate" e.g. more urgent.

We will develop a more nuanced approach to handling plan selection and intention scheduling by assigning weights (to plans and events). These will be encoded by reaction rules with weights using probabilistic bigraphs [63] that export Discrete-time Markov chains (DTMCs). As we export explicit transition systems, this approach can support many probabilistic logics such as PCTL [64] found in PRISM. We have begun preliminary work in this direction [65].

Our current encoding of BDI agents is limited to a self-static environment, i.e. the environment changes only when the agent changes it. We plan to develop a self-dynamic environment and will extend the mechanism of failure recovery to allow re-selection of previously failed plans. This will not only increase the persistence of an agent, but also increase the likelihood of success by taking advantage of environmental changes.

Finally, we also plan to address the problems of multi-agent systems. While it can be tempting to model the multi-agent system in an interleaved fashion (i.e. treating each agent as a thread within a system), a true multi-agent system should support true concurrency (e.g. one agent executes an action while another agent is performing another action). Recently, there is a promising work on true concurrency in BDI agent systems [66], which may be helpful to achieve this goal. Recent work [67] has also addressed scenarios where autonomous agents collaborate with humans to achieve a shared goal.

11. Conclusion

Rational agents, such as Belief-Desire-Intention (BDI) agents, will play a key role in future autonomous systems and it is essential we can reason about their behaviours, and provide early, i.e. design-time, indications of potential problems, e.g. deadlocks causes by shared resources.

We have presented a framework, based on Milner's bigraphs, for modelling and verifying BDI agents specified in the Can language. We believe this is the first *executable semantics* of Can, allowing verification of abstract agent programs, rather than verification based on a specific implementation of Can. The use of four perspectives in the bigraph model: Belief, Desire, Intention and Plan, helps is to separate concerns in the encoding and offers a clear visualisation of the resulting model.

The two key functions are the syntax encoding $[\![\cdot]\!]$ and $[\![\![\cdot]\!]\!]$ that enables the behavioural encoding. The former has the added feature of introducing event

indices for plans, which decreases errors and aids search. The latter is a bridge between agent-level and intention-level steps. Bigraph parallelism indicates how the belief base is the environment for reduction and conditional bigraphs allow us to prioritise reaction rules, which simplifies the encoding.

We have shown that the encoding of CAN agents in bigraphs is faithful by proving any CAN step is captured by a finite sequence of bigraph reaction rules, and we have shown shown the approach is practical through three example UAV applications. In each case, generating and verifying the model took no more than a few minutes.

This work has also highlighted many interesting features of the current semantics of CAN, such as the inability to distinguish between the success and failure of an intention and lack of meta-level reasoning, and it lays the foundation for future modelling work. We envisage an extended model (and hence extended semantics) that features probabilistic choice and dynamic environments – allowing quantitative model checking of agent programs.

Acknowledgements

This work is supported by the Engineering and Physical Sciences Research Council, under PETRAS SRF grant MAGIC (EP/S035362/1) and S4: Science of Sensor Systems Software (EP/N007565/1).

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, and J.-J. C. Meyer, "Agent programming in 3APL," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 4, pp. 357–401, 1999.

[4] M. Dastani, "2APL: a practical agent programming language," *Autonomous agents and multi-agent systems*, vol. 16, no. 3, pp. 214–248, 2008.

[5] R. Bordini, J. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason.* John Wiley & Sons, 2007, vol. 8.

[6] S. Sardina, L. d. Silva, and 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] S. S. Benfield, J. Hendrickson, and D. Galanti, "Making a strong business case for multiagent technology," in *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent systems*. ACM, 2006, pp. 10–15.

[8] L. Braubach, A. Pokahr, and W. Lamersdorf, "Negotiation-based patient scheduling in hospitals," in *Advanced Intelligent Computational Technologies and Decision Support Systems*, 2014, pp. 107–121.

[9] S. McArthur, E. Davidson, V. Catterson, A. Dimeas, N. Hatziargyriou, F. Ponci, and T. Funabashi, "Multi-agent systems for power engineering applications – part i: Concepts, approaches, and technical challenges," vol. 22, no. 4. IEEE, 2007, pp. 1743–1752.

[10] G. Brat, E. Denney, D. Giannakopoulou, J. Frank, and A. Jonsson, "Verification of autonomous systems for space applications," in *Proceedings of IEEE Aerospace Conference*, 2006.

[11] L. Lestingi, M. Askarpour, M. M. Bersani, and M. Rossi, "Formal verification of human-robot interaction in healthcare scenarios," in *Proceedings of International Conference on Software Engineering and Formal Methods*. Springer, 2020, pp. 303–324.

[12] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "Verifying multiagent programs by model checking," *Autonomous Agents and Multiagent Systems*, vol. 12, no. 2, pp. 239–256, 2006.

[13] G. J. Holzmann and W. S. Lieberman, *Design and validation of computer protocols*. Prentice hall Englewood Cliffs, 1991, vol. 512.

[14] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[15] L. A. Dennis, B. Farwer, R. H. Bordini, and M. Fisher, "A flexible framework for verifying agent programs," in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 2008, pp. 1303–1306.

[16] G. Brat, K. Havelund, S. Park, and W. Visser, "Model checking programs," in *Proceedings of IEEE International Conference on Automated Software Engineering*. IEEE, 2000, pp. 3–11.

[17] R. Milner, "Bigraphs and their algebra," *Electronic Notes in Theoretical Computer Science*, vol. 209, pp. 5–19, 2008.

[18] M. Bundgaard and V. Sassone, "Typed polyadic pi-calculus in bigraphs," in *Proceedings of ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2006, pp. 1–12.

[19] M. Sevegnani and M. Calder, "BigraphER: rewriting and analysis engine for bigraphs," in *Proceedings of International Conference on Computer Aided Verification*. Springer, 2016, pp. 494–501.

[20] B. Archibald, C. Muffy, and M. Sevegnani, "Conditional bigraphs," in *International Conference on Graph Transformation*. Springer, 2020, pp. 3–19.

[21] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott, "Maude manual (version 3.0)," *SRI International*, 2020.

[22] S. Benford, M. Calder, T. Rodden, and M. Sevegnani, "On lions, impala, and bigraphs: Modelling interactions in physical/virtual spaces," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 23, no. 2, pp. 1–56, 2016.

[23] S. Sardina and L. Padgham, "A BDI agent programming language with failure handling, declarative goals, and planning," *Autonomous Agents and Multi-Agent Systems*, pp. 18–70, 2011.

[24] R. Milner, *The space and motion of communicating agents*. Cambridge University Press, 2009.

[25] J. Meseguer, "Twenty years of rewriting logic," *J. Log. Algebraic Methods Program.*, vol. 81, no. 7-8, pp. 721–781, 2012. [Online]. Available: https://doi.org/10.1016/j.jlap.2012.06.003

[26] M. Calder and M. Sevegnani, "Modelling IEEE 802.11 CSMA/CA RTS/CTS with stochastic bigraphs with sharing," *Formal Aspects of Computing*, vol. 26, no. 3, pp. 537–561, 2014.

[27] G. D. Plotkin, "A structural approach to operational semantics," in *Lecture Notes, Aarhus University Denmark*, 1981.

[28] B. Logan, J. Thangarajah, and 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.

[29] M. Xu, K. McAreavey, K. Bauters, and W. Liu, "Intention interleaving via classical replanning," in *Proceedings of International Conference on Tools with Artificial Intelligence*, 2019, pp. 85–92.

[30] B. Archibald, M. Calder, M. Sevegnani, and M. Xu, "Modelling and verifying BDI agents with bigraphs – models," Jan. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4472541

[31] J. F. Groote, "Transition system specifications with negative premises," *Theoretical Computer Science*, vol. 118, no. 2, pp. 263–299, 1993.

[32] R. J. van Glabbeek, "The meaning of negative premises in transition system specifications ii," *The Journal of Logic and Algebraic Programming*, vol. 60, pp. 229–258, 2004.

[33] E. M. Clarke and 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.

[34] P. D. Mosses, "Modular structural operational semantics," *J. Log. Algebraic Methods Program.*, vol. 60-61, pp. 195–228, 2004. [Online]. Available: https://doi.org/10.1016/j.jlap.2004.03.008

[35] J. Harland, D. N. Morley, J. Thangarajah, and N. Yorke-Smith, "An operational semantics for the goal life-cycle in BDI agents," *Autonomous agents and multi-agent systems*, vol. 28, no. 4, pp. 682–719, 2014.

[36] ——, "Aborting, suspending, and resuming goals and plans in BDI agents," *Autonomous Agents and Multi-Agent Systems*, vol. 31, no. 2, pp. 288–331, 2017.

[37] B. Archibald, K. Burns, C. McCreesh, and M. Sevegnani, "Practical bigraphs via subgraph isomorphism," in *27th International Conference on Principles and Practice of Constraint Programming*, 2021.

[38] E. A. Emerson, "Temporal and modal logic," in *Formal Models and Semantics.* Elsevier, 1990, pp. 995–1072.

[39] L. A. Dennis, "Gwendolen semantics: 2017," *Technical Report ULCS-17-001, University of Liverpool*, 2017.

[40] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini, "Model checking agent programming languages," *Automated software engineering*, vol. 19, no. 1, pp. 5–63, 2012.

[41] L. A. Dennis, "The MCAPL framework including the agent infrastructure layer and agent Java PathFinder," *The Journal of Open Source Software*, 2018.

[42] L. Dennis, M. Fisher, M. Slavkovik, and M. Webster, "Formal verification of ethical choices in autonomous systems," *Robotics and Autonomous Systems*, vol. 77, pp. 1–14, 2016.

[43] K. V. Hindriks, F. S. De Boer, W. Van Der Hoek, and J.-J. C. Meyer, "Agent programming with declarative goals," in *Proceedings of International Workshop on Agent Theories, Architectures, and Languages.* Springer, 2000, pp. 228–243.

[44] L. A. Dennis, M. Fisher, and M. Webster, "Two-stage agent program verification," *Journal of Logic and Computation*, vol. 28, no. 3, pp. 499–523, 2018.

[45] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proceedings of International conference on computer aided verification.* Springer, 2011, pp. 585–591.

[46] P. Izzo, H. Qu, and S. M. Veres, "A stochastically verifiable autonomous control architecture with reasoning," in *Proceedings of IEEE Conference on Decision and Control.* IEEE, 2016, pp. 4985–4991.

[47] L. A. Dennis, M. Fisher, N. K. Lincoln, A. Lisitsa, and S. M. Veres, "Practical verification of decision-making in agent-based autonomous systems," *Automated Software Engineering*, vol. 23, no. 3, pp. 305–359, 2016.

[48] M. Wooldridge, M.-P. Huget, M. Fisher, and S. Parsons, "Model checking for multiagent systems: The MABLE language and its applications," *International Journal on Artificial Intelligence Tools*, vol. 15, no. 02, pp. 195–225, 2006.

[49] S. Shapiro, Y. Lespérance, and H. J. Levesque, "The cognitive agents specification language and verification environment for multiagent systems," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, 2002, pp. 19–26.

[50] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker," *Electronic Notes in Theoretical Computer Science*, vol. 71, pp. 162–187, 2004.

[51] M. B. Van Riemsdijk, F. S. De Boer, M. Dastani, and J.-J. C. Meyer, "Prototyping 3APL in the Maude term rewriting language," in *International Workshop on Computational Logic in Multi-Agent Systems.* Springer, 2006, pp. 95–114.

[52] T. T. Doan, Y. Yao, N. Alechina, and B. Logan, "Verifying heterogeneous multi-agent programs," in *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, 2014, pp. 149–156.

[53] S.-S. T. Jongmans, K. V. Hindriks, and M. B. Van Riemsdijk, "Model checking agent programs by using the program interpreter," in *International Workshop on Computational Logic in Multi-Agent Systems.* Springer, 2010, pp. 219–237.

[54] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge, "State-space reduction techniques in agent verification," in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 2004, pp. 896–903.

[55] R. H. Bordini, M. Fisher, M. Wooldridge, and W. Visser, "Property-based slicing for agent verification," *Journal of Logic and Computation*, vol. 19, no. 6, pp. 1385–1425, 2009.

[56] M. Winikoff, L. Dennis, and M. Fisher, "Slicing agent programs for more efficient verification," in *International Workshop on Engineering Multi-Agent Systems*. Springer, 2018, pp. 139–157.

[57] S.-S. T. Jongmans, K. V. Hindriks, and M. B. Van Riemsdijk, "State space reduction for model checking agent programs," in *International Workshop on Programming Multi-Agent Systems*. Springer, 2011, pp. 133–151.

[58] R. Milner, "Pure bigraphs: structure and dynamics," *Information and computation*, vol. 204, no. 1, pp. 60–122, 2006.

[59] O. H. Jensen, "Mobile processes in bigraphs," Ph.D. dissertation, University of Aalborg, 2006.

[60] M. Sevegnani and E. Pereira, "Towards a bigraphical encoding of actors," in *Proceedings of International Workshop on Meta Models for Process Languages*, 2014.

[61] A. Mansutti, M. Miculan, and M. Peressotti, "Multi-agent systems design and prototyping with bigraphical reactive systems," in *IFIP international conference on distributed applications and interoperable systems*. Springer, 2014, pp. 201–208.

[62] A. T. E. Dib and Z. Sahnoun, "Model checking of multi-agent system architectures using BigMC," in *Proceedings of Federated Conference on Computer Science and Information Systems*, 2015, pp. 1717–1722.

[63] B. Archibald, M. Calder, and M. Sevegnani, "Probabilistic bigraphs," *Submitted for publication*, 2021, preprint at https://arxiv.org/abs/2105.02559.

[64] A. Bianco and L. De Alfaro, "Model checking of probabilistic and nondeterministic systems," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 1995, pp. 499–513.

[65] B. Archibald, M. Calder, M. Sevegnani, and M. Xu, "Probabilistic BDI agents: Actions, plans and intentions." in *Proceedings of 19th Intl. Conference on Software Engineering and Formal Methods*, 2021.

[66] L. De Silva, "An operational semantics for true concurrency in BDI agent systems," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 7119–7126.

[67] B. Archibald, M. Calder, M. Sevegnani, and M. Xu, "Observable and attention-directing BDI agents for human-autonomy teaming," in Proceedings Third Workshop on *Formal Methods for Autonomous Systems*, ser. Electronic Proceedings in Theoretical Computer Science, M. Farrell and M. Luckcuck, Eds., vol. 348. Open Publishing Association, 2021, pp. 167–175.