Encoding BDI Syntax with Theories in Event-B

Mengwei Xu[1(✉)], Peter Rivière[2], Toshiaki Aoki[2], Marie Farrell[3], Yamine Aït Ameur[4], Neeraj Kumar Singh[4], and Guillaume Dupont[4]

[1] Newcastle University, Newcastle upon Tyne, UK
mengwei.xu@newcastle.ac.uk
[2] Japan Advanced Institute of Science and Technology, Nomi, Japan
{priviere,toshiaki}@jaist.ac.jp
[3] The University of Manchester, Manchester, UK
marie.farrell@manchester.ac.uk
[4] IRIT, Toulouse National Polytechnique Institute - CNRS, Université de Toulouse, Toulouse, France
{yamine,nsingh}@enseeiht.fr, guillaume.dupont@toulouse-inp.fr

Abstract. The Belief–Desire–Intention (BDI) paradigm is a popular framework in the development of autonomous systems. However, assuring the correct design of BDI agents remains difficult: existing modelling formalisms often require ad-hoc encodings of BDI agents that can be difficult to validate, maintain, and reason about. This paper focuses on modelling the syntax of BDI agents and shows how algebraic modelling in Event-B theories (e.g. inductive data types and polymorphic constructors) yields a faithful, compact, and reusable encoding of BDI syntax. Even without committing to the full BDI semantics, the encoding already supports useful reasoning, including belief entailment and belief-based invariant checking, and provides a path towards a future BDI semantic encoding via operators in theories and machine events in Event-B.

Keywords: BDI agents · Algebraic modelling · Event-B theories

1 Introduction

The Belief–Desire–Intention (BDI) paradigm [15] has been a popular framework to design and develop autonomous systems that make decisions and execute actions without human intervention, e.g. in robotics [12]. Rooted in Bratman's philosophical work [10], the (B)eliefs represent what the agent knows, the (D)esires what the agent wants to bring about, and the (I)ntentions those desires that the agent has committed to act upon. For instance, a robot may maintain

This work was partially supported by the Royal Academy of Engineering, EPSRC grant EP/Y001532/1, ANR-19-CE25-0010 *EBRP:EventB-Rodin-Plus* project and JST, CREST Grant Number JPMJCR23M1.

F. Ishikawa and A. Cunha (Eds.): ABZ 2026, LNCS 16579, pp. 210–228, 2026.
https://doi.org/10.1007/978-3-032-26752-8_13

beliefs about its current position and nearby hazards, desires such as reaching a particular inspection location, and intentions of the current navigation plan.

However, assuring the correct design of BDI agents is challenging due to: (i) modelling expressiveness and (ii) analysis scalability. The former is limited because many existing modelling formalisms are effective for the domains that they target (e.g. communication systems) but are not tailored to BDI agents. Using these modelling languages often leads to ad-hoc encodings that are difficult to validate or reuse. The latter persists because model checking—a popular verification technique for BDI agents—suffers from state space explosion [13] as agent complexity increases for realistic scenarios, despite its high automation.

Significant work on formal modelling and analysis of BDI agents is surveyed in [26]. Many approaches adapt modelling formalisms originally intended for their target domains to BDI agents through ad-hoc encodings. For example, one of the earliest approaches [6] employs Promela [23], a modelling language for communication-based systems, to model and analyse BDI agents with the Spin model checker [22]. Similarly, the work [5,19] use re-writing logics either term-based (e.g. Maude [14]) or graph-based (e.g. Bigraphs [27]) to model BDI agents. The work [17] even avoids the modelling process by employing the program model checker Java PathFinder [20] directly on Java implementations of BDI languages e.g. Gwendolen [16]. While useful, the exhaustive state exploration in model checking leaves them vulnerable to state space explosion. To address this, recent proof-based approaches specify BDI agents in proof assistants (e.g. Isabelle/HOL [28]) using algebraic data types [24,31] on two simple BDI agents (GOAL [21] and SimpleBDI [18]) where the BDI plans only have actions (e.g. no sub-goals). In contrast, we use Event-B theories to model BDI syntax algebraically without simplification, while retaining Event-B's state-based style for future work on modelling the configuration-transition BDI semantics.

In this paper, we present an algebraic modelling approach to encode the syntax of a fully fledged BDI programming language, specified in the Conceptual Agent Notation (Can), using Event-B formalism [1]. Can is chosen as it includes advanced BDI agent behaviours such as declarative goals, concurrency, and failure recovery. The same modelling techniques here would apply to other BDI languages. We access algebraic modelling through Event-B theories [11] to define inductive data types and polymorphic constructors that faithfully mirror the vocabulary of the Can syntax as written. The encoding, as implemented in Rodin [2], is both total and injective with respect to the original Can syntax. The outcome is a suite of self-contained Event-B theories that serve as a foundation for specifying concrete BDI agent programs, keeping models reusable.

In Sect. 2, we introduce the preliminaries for the Can language with a running example and Event-B formalism; Sect. 3 presents algebraic modelling of Can syntax in Event-B theories; Sect. 4 shows possible reasoning mechanisms for our modelling without attaching to any specific BDI semantics; Sect. 5 discusses related work and Sect. 6 concludes and outlines future work.

2 Preliminaries

2.1 CAN Syntax

We give an overview of the syntax of the CAN language from [29,30] to which we refer for full details and a running example of CAN agents. A CAN agent consists of a belief base, $\mathcal{B}$, a plan library, Π, and an action description library, Λ. The syntax of a CAN agent is constructed by three main types of predicate symbols, namely event predicate symbols, e, belief predicate symbols, b, and action predicate symbols, act. Together with terms, they form the events, beliefs, and actions. Standard first-order terms and free/bound variables are used. For example, terms (resp. vector terms) in CAN are denoted as t (resp. $\mathbf{t}$), and we can write $e(\mathbf{t})$, $b(\mathbf{t})$, and $act(\mathbf{t})$ to denote events, beliefs, and actions, respectively.

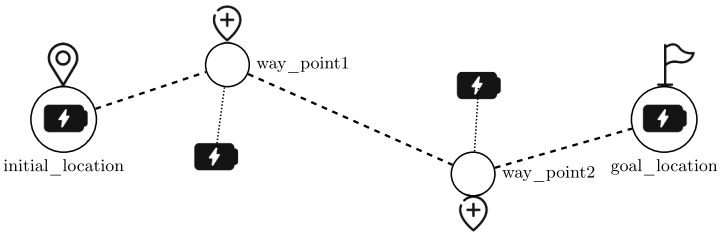

Fig. 1. Rover navigation and charging scenario with two way points.

The *belief base*, $\mathcal{B}$, represents the current beliefs of the agent. If b is a predicate symbol, and $t_1, \ldots, t_n$ are terms, then $b(t_1, \ldots, t_n)$ or $b(\mathbf{t})$ is a belief atom. Such an atom is *ground* if all terms in $\mathbf{t}$ are ground terms, i.e. it contains no variables. The belief base $\mathcal{B}$ is defined as a set of ground belief atoms.

A *plan library*, Π, contains the operational procedures and is a finite collection of plans of the form $e(\mathbf{t}) : \varphi(\mathbf{x_t}, \mathbf{y}) \leftarrow P(\mathbf{x_t}, \mathbf{y})$. Here, $e(\mathbf{t})$ is the triggering event of this plan, $\varphi(\mathbf{x_t}, \mathbf{y})$ is the context condition, and $P(\mathbf{x_t}, \mathbf{y})$ is the plan-body. The triggering event, $e(\mathbf{t})$, specifies *why* the plan is triggered, while the context condition, $\varphi(\mathbf{x_t}, \mathbf{y})$, determines *when* the plan-body P is applicable. The variable $\mathbf{x_t}$ denotes all of the free variables in the terms $\mathbf{t}$, and variables $\mathbf{y}$ are those free variables that do not appear in the triggering event but are introduced in the context condition, generally to bind objects that are to be used in the plan-body, $P(\mathbf{x_t}, \mathbf{y})$. The context condition $\varphi(\mathbf{x_t}, \mathbf{y})$ is a belief formula, built from belief atoms using the standard logical connectives, with syntax $\varphi ::= b(\mathbf{t}) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$. Unlike the belief base, which contains only ground belief atoms, a context condition may contain free variables. Such variables are interpreted semantically with respect to the current grounded belief base.

Events may arise either from the external environment or from within the execution of a plan-body. We write E^e for the set of external events. Events

```
1  // Initial belief base
2  at(init_loc), power(high), next(init_loc,WP1), next(WP1,WP2), next(WP2,goal_loc)
3  nearest(init_loc,init_loc_charge), nearest(init_loc_charge, init_loc)
4  nearest(WP1,WP1_charge), nearest(WP1_charge,WP1)
5  nearest(WP2,WP2_charge), nearest(WP2_charge, WP2),
6  nearest(goal_loc,goal_loc_charge), nearest(goal_loc_charge, goal_loc)
```

(a) Belief base.

```
1  // Initial external events
2  start(mission)
3  // Plan library.
4  start(mission):at(X)<-goal(at(goal_loc), navigate(goal_loc),power(empty)∧¬at(goal_loc))
6  navigate(Z):at(goal_loc)<-stop(Z)
7  navigate(Z):at(X)∧next(X,Y)∧¬power(empty)<-movement(X,Y);navigate(Z)
8  navigate(Z):at(X)∧nearest(X,C)∧power(low) <-
9       charge_locate(X,C); charge(C); charge_locate(C,X); navigate(Z)
10 movement(X,Y): power(high) <- move_high(X,Y)
11 movement(X,Y): power(low) <- move_low(X,Y)
```

(b) Plan library.

```
1  // Action Description Library
2  move_high(X,Y):at(X)∧next(X,Y)∧power(high)<- ⟨{at(Y), power(low)},{at(X), power(high)}⟩
3  move_low(X,Y):at(X)∧next(X,Y)∧power(low)<- ⟨{at(Y), power(empty)},{at(X), power(low)}⟩
4  charge_locate(X,C): at(X) ∧ nearest(X,C) <- ⟨{at(C)},{at(X)}⟩
5  charge(C): at(C) ∧ power(low) <- ⟨{power(high)},{power(low)}⟩
6  stop(Z): ⊤ <- ⟨∅, ∅⟩
```

(c) Action description library

Fig. 2. BDI agent design and plan library in the rover scenario

occurring as part of a plan-body (which will be introduced next) are called *sub-events* or *internal events*. For presentation convenience, we may omit arguments of terms when they do not confuse, e.g. e in place of $e(\mathbf{t})$.

By convention e.g. in [7], the user-defined plan-body, P, may be referred to as the *program* or *agent program* and has the following syntax:

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

where act is an action, $?\varphi$ a test for φ entailment in the belief base, $+b$ and $-b$ represent belief addition and deletion respectively, and e is a sub-event (i.e. internal event). To execute a sub-event, a corresponding plan is selected and the plan-body is added in place of the event. In this way, plans can be nested. Actions, act, (from the action description library Λ) take the form $act = \psi \leftarrow \langle \phi^+, \phi^- \rangle$, where ψ is the pre-condition, and ϕ^+ and ϕ^- are the addition and deletion sets of belief atoms. There are also composite programs $P_1; P_2$ for sequence and $P_1 \parallel P_2$ for interleaved concurrency. Finally, a declarative goal, $goal(\varphi_s, e, \varphi_f)$, expresses that the state, φ_s, should be achieved through addressing the event, e, failing if φ_f is true, and re-trying if neither φ_s nor φ_f are true.

2.2 A Running Example: Rover Navigation and Charging

Figure 1 presents a rover navigation and charging scenario (adapted from [8,9, 12]). The rover must reach `goal_loc` from `init_loc` by traversing a sequence

Table 1. Structure of Event-B contexts, machines, and theories.

(a) Context	(b) Machine	(c) Theory
CONTEXT Ctx	MACHINE M	THEORY Th
SETS s	SEES Ctx	IMPORT $Th1, \ldots$
CONSTANTS c	VARIABLES x	TYPE PARAMETERS E, F, ...
AXIOMS $A(s, c)$	INVARIANTS $I(x)$	DATA TYPES
THEOREMS $T_{ctx}(s, c)$	EVENTS	**Type1**$(E, \ldots)$
END	EVENT evt	constructors
	ANY α	**cstr1**$(p_1 \colon T_1, \ldots)$
	WHERE $G(x, a)$	AXIOMATIC DEFINITIONS
	$x :\mid BAP(\alpha, x, x')$	TYPES $A_1, \ldots$
	END	OPERATORS
	...	**AOp2** <nature> $(p_1 \colon T_1, \ldots) \colon T_r$
	END	well−definedness $WD(p_1, \ldots)$
		AXIOMS $A_1, \ldots$
		END

of waypoints (WPs), and it must detour to a nearby charging station when the power is low. To capture this rover scenario in CAN, we instantiate a BDI agent as follows. The initial belief base (Fig. 2a) captures the topology using belief predicates `at(X)`, `next(X,Y)`, and `nearest(X,C)`. The rover's power level is modelled using three mutually exclusive belief atoms: `power(high)`, `power(low)`, and `power(empty)`, initially set to `power(high)` on line 2.

The rover's mission is initiated by the external `start(mission)` event on line 2 of Fig. 2b. This event is addressed by the plan (line 4) through a declarative goal to achieve `at(goal_loc)` via the internal event, `navigate(goal_loc)`. The goal succeeds when the rover reaches the goal location, and fails when its power is empty and it is not at the goal location (i.e. `power(empty)`$\wedge\neg$`at(goal_loc)`).

Navigation proceeds in discrete steps. When the rover is not in the empty-power state, it advances along the waypoint chain using `next(X,Y)` (line 7). Rather than directly calling a movement action, the plan delegates to an internal event, `movement(X,Y)`, which branches depending on the current power status (lines 10–11). The corresponding action description (lines 2–3 of Fig. 2c) updates the location to `at(Y)` and degrades the power from high to low or low to empty.

When the rover is in the low-power mode, it can also take a charging detour (lines 8–9). Using `nearest(X,C)`, the agent relocates to the nearest charging station `C` via `charge_locate(X,C)`, which consumes negligible power, performs `charge(C)` to restore `power(high)`, and then returns to the original waypoint `X` via `charge_locate(C,X)` before resuming `navigate(Z)`. We note that, in low-power situations, the two `navigate(Z)` plans (lines 7–8) may both be applicable,

and one plan is selected non-deterministically by the agent. Once `at(goal_loc)` holds, the termination plan triggers `stop(Z)` (line 6), i.e. doing nothing.

Throughout this paper, we will use this running example to illustrate how the Event-B theories in the Rodin platform can encode BDI syntax.

2.3 Event-B

We briefly recall the relevant Event-B notions we use in this paper [1]. A *context* (Table 1(a)) provides static modelling via carrier sets, s, constants, c, axioms, $A(s, c)$ and theorems, $T_{ctx}(s, c)$. A *machine* (Table 1(b)) provides a state-based model via variables, x, invariants, $I(x)$, and events (with before–after updates). Proof obligations are generated by the Rodin Platform to ensure invariant preservation, among other things. To support algebraic modelling beyond core set theory and first-order logic, we use Event-B *theories* [3,11]. A theory (Table 1(c)) can import others and define polymorphic data types and operators, either by constructors or axiomatically. We will provide further details on Event-B theories (and, later, machines and events) where required for encoding BDI agents.

```
1  THEORY TermsTheory
2  AXIOMATIC DEFINITIONS
3   TYPES: Terms
4   OPERATORS
5     PointTerms: ℙ(Terms)
6     TwoAryVectorTerms: ℙ(Terms)
7     ...
8     NAryVectorTerms: ℙ(Terms)
9     TwoAryVectorFunction(P1: Terms, P2: Terms) : Terms
10      well−definedness:
11          P1 ∈ PointTerms
12          P2 ∈ PointTerms
13    ...
14    NAryVectorFunction(P1: Terms, P2: Terms, ···, Pn: Terms) : Terms
15      well−definedness:
16          P1 ∈ PointTerms
17          P2 ∈ PointTerms
18          ...
19          Pn ∈ PointTerms
20  AXIOMS
21     axm1: partition(Terms, PointTerms, TwoAryVectorTerms, ··· , NAryVectorTerms)
22     axm2: ∀P1, P2 · P1 ∈ PointTerms ∧ P2 ∈ PointTerms
23           ⇒ TwoAryVectorFunction(P1, P2) ∈ TwoAryVectorTerms
24     axm3: ∀P · P ∈ TwoAryVectorTerms
25           ⇒ (∃P1, P2 · P1 ∈ PointTerms ∧ P2 ∈ PointTerms
26                                  ∧P = TwoAryVectorFunction(P1, P2))
27       ...
```

Fig. 3. Event-B theory for terms

3 Encoding the Syntax of CAN in Event-B

In this section, we encode CAN syntax in Event-B (theories and contexts in Rodin). We treat predicate symbols (belief, event, action) and terms as primitive alphabets that are introduced axiomatically, and encode the inductive fragments (e.g. plan-bodies) using recursive data-type definitions.

```
1  CONTEXT TermsRoverNavigation
2  CONSTANTS init_loc, WP1, WP2, goal_loc, mission, empty, low, high,
3            init_loc_charge, WP1_loc_charge, WP2_loc_charge, goal_loc_charge
4  AXIOMS
5    axm1: partition(PointTerms, {init_loc}, {WP1}, {WP2}, {goal_loc}, {mission}, {empty}
6                    {low}, {high}, {init_loc_charge}, {WP1_loc_charge}, {WP2_loc_charge},
7                    {goal_loc_charge})
```

Fig. 4. Context for terms in rover scenario.

```
1  THEORY PredicateSymbolsTheory
2  AXIOMATIC DEFINITIONS
3  TYPES: Event_predicate_symbols, Belief_predicate_symbols, Action_predicate_symbols
```

Fig. 5. Event-B theory for predicate symbols.

3.1 Terms

We begin with first-order terms including point terms (i.e. non-vector terms) and vector terms. Vector terms apply an n-ary function, f, ($n \geq 2$) to point terms, written as $f(t_1, \ldots, t_n)$ (or $\mathbf{t}$). For CAN, it suffices to restrict the arguments to point terms, without loss of generality, to illustrate our approach. We note that, although first-order logic includes variable terms, we have captured them in Event-B via the quantification and set comprehension over the domain of *ground* terms. Hence, we do not introduce variable terms as a separate syntactic category; instead, we work with the set of ground terms, defined as follows:

$$\begin{aligned}
\langle \text{Terms} \rangle &::= \langle \text{Point-Terms} \rangle \mid \langle \text{Vector-Terms} \rangle \\
\langle \text{Vector-Terms} \rangle &::= \langle \text{N-Ary-Function} \rangle (\langle \text{Point-Terms} \rangle_1, \ldots, \langle \text{Point-Terms} \rangle_n) \\
\langle \text{N-Ary-Function} \rangle &::= f \text{ where } f \text{ is an } n\text{-ary function}, n \geq 2, n \in \mathbb{N}
\end{aligned}$$

The axiomatic definition in Fig. 3 models ⟨Terms⟩ with the user-defined type, $Terms$, as the collection of all grounded terms (line 3). A list of nullary operators e.g. $PointTerms$ and $TwoAryVectorTerms$ in OPERATORS (lines 4–19), is introduced with type $\mathbb{P}(Terms)$ to denote particular subsets of $Terms$. Here, a nullary operator is a constant that encodes an entity with an explicit type.

Meanwhile, **axm1** (line 21) specifies that the named subsets form a partition of the whole domain of terms. The $TwoAryVectorFunction$ operator (lines 9–12) constructs two-ary vector terms from two point terms. Its well-definedness ensures that both arguments lie in $PointTerms$. Both **axm2** and **axm3** formally describe the well-definedness between $PointTerms$ and $TwoAryVectorTerms$. The **axm2** specifies that applying $TwoAryVectorFunction$ to any two point terms yields an element of $TwoAryVectorTerms$. And **axm3** goes in the other direction: any element in $TwoAryVectorTerms$ arises from some pair of point terms. Similar explanations (omitted) can be given for $NAryVectorFunction$.

To provide domain-specific terms, we can import this theory and apply it to any context within any Rodin project. Recall that, in the Rover navigation scenario, the terms include all locations, `mission` to start mission, and `empty` to `high` for the battery status. To encode them in Event-B, we can have the

context shown in Fig. 4. The name of this context is given in the CONTEXT clause (line 1). It first declares all constants e.g. *init_loc* in the CONSTANTS clause. Then **axm1** (lines 5–7) assigns them as the only elements of *PointTerms*. This yields a finite, grounded set of point terms for the rover navigation scenario. We will show how to construct vector terms from point terms using the $TwoAryVectorFunction$ operator when encoding the belief base.

3.2 Predicate Symbols

We have modelled terms, but predicates consist of both predicate symbols and terms. We now encode predicate symbols by introducing their types (no operators or axioms) in a separate axiomatic definition (see Fig. 5), which can be imported for domain-specific extensions. For the rover scenario, belief symbols include `at`, `next`, `nearest`, and `power`; event symbols have `start`, `navigate`, and `movement`; and action symbols contain `move_high`, `move_low`, `charge_locate`, `charge`, and `stop`. These are categorised in a context (Fig. 6).

```
1  CONTEXT PredicateSymbolsRoverNavigation
2  CONSTANTS at, next, nearest, power, start, navigate, , movement,
3            move_high, move_low, charge_locate, charge, stop,
4  AXIOMS
5    axm1: partition(Belief_predicate_symbols, {at}, {next}, {nearest}, {power})
6    axm2: partition(Event_predicate_symbols, {start}, {navigate}, {movement})
7    axm3: partition(Action_predicate_symbols, {move_high}, {move_low}, {charge_locate},
8                    {charge}, {stop})
```

Fig. 6. Context for predicate symbols in rover scenario.

```
1  THEORY InitialBeliefBaseExternalEventsTheory
2  IMPORT TermDefinition, PredicateSymbolsDefinition
3  AXIOMATIC DEFINITIONS
4   OPERATORS
5     Initial_belief_base: ℙ(Belief_predicate_symbols × Terms)
6     Initial_external_events: ℙ(Event_predicate_symbols × Terms)
```

Fig. 7. Theory for initial beliefs base and external events.

```
1  CONTEXT InitialBeliefBaseExternalEventsRoboticCleaning
2  AXIOMS
3    axm1: Initial_belief_base = {
4            (at ↦ init_loc), (power ↦ high), (next ↦ TwoAryVectorFunction(init_loc, WP1))
5            (next ↦ TwoAryVectorFunction(WP1, WP2)),
6            (next ↦ TwoAryVectorFunction(WP2, goal_loc)),
7            (nearest ↦ TwoAryVectorFunction(init_loc, init_charge)),
8            (nearest ↦ TwoAryVectorFunction(init_charge, init_loc)),
9            (nearest ↦ TwoAryVectorFunction(WP1, WP1_charge)),
10           (nearest ↦ TwoAryVectorFunction(WP1_charge, WP1)),
11           (nearest ↦ TwoAryVectorFunction(WP2, WP2_charge)),
12           (nearest ↦ TwoAryVectorFunction(WP2_charge, WP2)),
13           (nearest ↦ TwoAryVectorFunction(goal_loc, goal_loc_charge)),
14           (nearest ↦ TwoAryVectorFunction(goal_loc_charge, goal_loc)}
15   axm2: Initial_external_events = {(start ↦ mission)}
```

Fig. 8. Initial belief base and external events for rover scenario.

3.3 Initial Belief Base and External Events

Building on the term and predicate symbol theories, we encode the initial belief base and external events as *typed constants* (nullary operators) in an axiomatic theory. Concretely, Fig. 7 declares *Initial_belief_base* with type $\mathbb{P}(Belief\_predicate\_symbols \times Terms)$ (and similarly for the initial external events). The theory only fixes these types; a separate context instantiates them for the rover case study in Fig. 8. For example, `next(init_loc, WP1)` is encoded as $(next \mapsto TwoAryVectorFunction(init\_loc, WP1))$ on line 4, and the sole external event `start(mission)` is encoded as $(start \mapsto mission)$ on line 15.

3.4 Agent Programs

$$
\begin{aligned}
\langle \text{UserP} \rangle &::= \langle \text{BasicP} \rangle \mid \langle \text{UserP} \rangle ; \langle \text{UserP} \rangle \mid \\
&\quad \langle \text{UserP} \rangle \| \langle \text{UserP} \rangle \mid \mathit{goal}(\langle \text{BeliefFormula} \rangle, \texttt{e}, \langle \text{BeliefFormula} \rangle) \\
\langle \text{BasicP} \rangle &::= \texttt{e} \mid \texttt{act} \\
\langle \text{BeliefFormula} \rangle &::= \texttt{b} \mid \neg \langle \text{BeliefFormula} \rangle \mid \langle \text{BeliefFormula} \rangle \wedge \langle \text{BeliefFormula} \rangle \mid \\
&\quad \langle \text{BeliefFormula} \rangle \vee \langle \text{BeliefFormula} \rangle \mid \top \mid \bot
\end{aligned}
$$

Fig. 9. Grammar for agent programs.

We now proceed to encode agent programs using inductive data types. Recall that the user-defined plan-body, P, in a plan, $\mathrm{e} : \varphi \leftarrow P$, is often called an agent program. The grammar for agent programs in Can is formally given in Fig. 9 where the grammar category $\langle \text{UserP} \rangle$ denotes plan-body P. $\langle \text{UserP} \rangle$ can be the basic building block $\langle \text{BasicP} \rangle$ including an internal event `e` or an action `act`. We have omitted the inclusion of $+b$, $-b$, and $?\varphi$ as they can be seen as special cases of actions. For example $?\varphi$ can be seen as $act : \varphi \leftarrow \langle \emptyset, \emptyset \rangle$ whereas $+b$ as $act : \top \leftarrow \langle \{b\}, \emptyset \rangle$. $\langle \text{UserP} \rangle$ can also be a declarative goal $\mathit{goal}(\langle \text{BeliefFormula} \rangle, e, \langle \text{BeliefFormula} \rangle)$. In a declarative goal, $\langle \text{BeliefFormula} \rangle$ represents the belief formula which is defined over a finite set of belief atoms, $\mathcal{A}t$, such that $\texttt{b} \in \mathcal{A}t$ using the standard logical connectives ($\neg$, $\wedge$, and $\vee$). The first belief formula represents the success state (i.e. φ_s) to achieve through addressing an internal event, e, failing when the second belief formula representing a failure condition (i.e. φ_f) holds, and retrying as long as neither of these two belief formulas is true. Finally, $\langle \text{UserP} \rangle$ can be composed in two ways: (i) $\langle \text{UserP} \rangle ; \langle \text{UserP} \rangle$ executing those two $\langle \text{UserP} \rangle$ in sequence and (ii) $\langle \text{UserP} \rangle \parallel \langle \text{UserP} \rangle$ executing those two $\langle \text{UserP} \rangle$ concurrently.

The grammar for $\langle \text{UserP} \rangle$ is, by nature, recursive, and also dependent on $\langle \text{BeliefFormula} \rangle$ (which is again recursively defined). To encode $\langle \text{UserP} \rangle$, we first employ the Event-B theory to model a belief formula with an inductive

```
1  THEORY BeliefFormulaTheory
2  DATATYPE
3    BeliefFormula(propositional_atoms)
4    CONSTRUCTORS
5      Belief_atom(atom : propositional_atoms)
6      Conjunctive_formula(
7          left_formula : BeliefFormula(propositional_atoms)
8          right_formula : BeliefFormula(propositional_atoms))
9      Disjunctive_formula(
10       left_formula : BeliefFormula(propositional_atoms)
11       right_formula : BeliefFormula(propositional_atoms))
12     Negated_formula(formula : BeliefFormula(propositional_atoms))
13     TruthFormula
14     FalseFormula
```

Fig. 10. Event-B theory for belief formula.

```
1  THEORY AgentProgramsTheory
2  IMPORTS BeliefFormulaTheory , TermsTheory , PredicateSymbolsTheory
3  DATATYPE
4    UserP
5    CONSTRUCTORS
6      BasicP_event_user(basic_program_event_user : Event_predicate_symbols × Terms)
7      BasicP_action_user(basic_program_action_user : Action_predicate_symbols × Terms)
8      Sequence_program_user(
9        head_program_user : UserP
10       tail_program_user : UserP)
11     Concurrency_program_user(
12       left_program_user : UserP
13       right_program_user : UserP)
14     Declarative_goal_user(
15       success_condition_user : BeliefFormula(Belief_predicate_symbols × Terms)
16       event_user : Event_predicate_symbols × Terms
17       failure_condition_user : BeliefFormula(Belief_predicate_symbols × Terms))
```

Fig. 11. Event-B theory for agent programs.

data type given in Fig. 10. The DATATYPE header (line 3) makes explicit that *BeliefFormula* is parameterised by a set of propositional atoms. The six polymorphic constructors (lines 4–14) show the recursive nature of the belief formulas. *Belief_atom* (line 5) forms an atomic formula. The constructors on lines 6–12 then encode logical conjunction, disjunction, and negation over sub-formulas. As a result, formulas are built from formulas inductively. Finally, $TruthFormula$ and $FalseFormula$ (lines 13–14) represent truth and falsehood.

With the defined data type of $BeliefFormula$, we can encode the agent programs in Fig. 11, namely *AgentProgramsTheory* where the data type $UserP$ is defined inductively. Here $UserP$ builds on the earlier $BeliefFormulaTheory$, $TermsTheory$, and *PredicateSymbolsTheory* by importing them (line 2). $UserP$ includes a list of constructors covering all cases from ⟨UserP⟩ in Fig. 9. For example, the constructor of $BasicP\_event\_user$ and $BasicP\_action\_user$ correspond to ⟨BasicP⟩ i.e. event and action. For example, to encode the action `charge(WP1)` to `charge` at the `WP1`, we can have $BasicP\_action\_user(charge \mapsto WP1)$ where $charge \in Action\_predicate\_symbols$ and $WP1 \in Terms$.

Similarly, compositional constructors including $Sequence\_program\_user$ and $Concurrency\_program\_user$ (lines 8–13) corresponds to ⟨UserP⟩; ⟨UserP⟩

```
THEORY PlanLibraryActionDescriptionLibraryTheory
IMPORT AgentProgramsTheory
AXIOMATIC DEFINITIONS
 OPERATORS
   plan_library : ℙ((Event_predicate_symbols × Terms)
                    ×(BeliefFormula(Belief_predicate_symbols × Terms) × UserP))
   action_description_library :
     ℙ((Action_predicate_symbols × Terms)
       ×(BeliefFormula(Belief_predicate_symbols × Terms)
       ×(ℙ(Belief_predicate_symbols × Terms) × ℙ(Belief_predicate_symbols × Terms))))
```

Fig. 12. Event-B theory for plan/action description library

```
CONTEXT PlanLibraryActionDescriptionLibraryContextRoverNavigation
EXTENDS TermsRoverNavigation , PredicateSymbolsRoverNavigation
CONSTANTS plan1 , plan2 , plan3 , plan4 , plan5 , plan6 ,
          act1 , act2 , act3 , act4 , act5
AXIOMS
  axm1 : plan_library = plan1 ∪ plan2 ∪ plan3 ∪ plan4 ∪ plan5 ∪ plan6
  axm2 : action_description_library = act1 ∪ act2 ∪ act3 ∪ act4 ∪ act5
  axm3 : plan1 = {triggering_event, context, plan_body, success_condition,
                  failure_condition, procedural_program·
                  triggering_event ∈ Event_predicate_symbols × Terms
                  ∧triggering_event = start ↦ mission
                  ∧context = Belief_atom(at ↦ X)
                  ∧success_condition = Belief_atom(at ↦ goal_loc)
                  ∧failure_condition = Conjunctive_formula(
                                       Belief_atom(power ↦ empty),
                                       Negated_formula(Belief_atom(at ↦ goal_loc)))
                  ∧procedural_program ∈ Event_predicate_symbols × Terms
                  ∧procedural_program = navigate ↦ goal_loc
                  ∧plan_body = Declarative_goal_user(success_condition,
                                                     procedural_program, failure_condition)
                  | triggering_event ↦ (context ↦ plan_body)
                  }
```

Fig. 13. Context of plan/action description library in rover scenario

and ⟨UserP⟩ ∥ ⟨UserP⟩ to construct sequential and interleaved agent programs over any legitimate agent program. Declarative goals are encoded via the constructor of *Declarative_goal_user*: a user-specified declarative goal is defined in terms of a success condition and a failure condition, both expressed using the *BeliefFormula* data type, along with the procedural program of an event in the middle with the type $Event\_predicate\_symbols \times Terms$.

3.5 Plan Library and Action Description Library

We now encode the plan library, Π, and the action description library, Λ. Recall that Π contains a set of plans, $e : \varphi \leftarrow P$, where e is the triggering event, φ the belief formula, and P is an agent program. The action library, Λ, is the set of actions, $act = \psi \leftarrow \langle \phi^+, \phi^- \rangle$, where ψ is the pre-condition (i.e. a belief formula), and ϕ^+ and ϕ^- are the addition and deletion sets of belief atoms.

Figure 12 axiomatically encodes the plan library and action description library using nullary operators. For example, the nullary operator *plan_library* has type:

```
1  OPERATORS
2   belief_entail <predicate>
3         (belief_base : ℙ(propositional_atoms),
4          formula : BeliefFormula(propositional_atoms))
5   recursive definition
6    case formula:
7      Belief_atom(atom) ⇒ atom ∈ belief_base
8      Conjunctive_formula(left, right) ⇒
9              belief_entail(belief_base, left) ∧ belief_entail(belief_base, right)
10     Disjunctive_formula(left, right) ⇒
11             belief_entail(belief_base, left) ∨ belief_entail(belief_base, right)
12     Negated_formula(f) ⇒ ¬belief_entail(belief_base, f)
13     TruthFormula ⇒ ⊤
14     FalseFormula ⇒ ⊥
```

Fig. 14. Belief entail operator.

$$\mathbb{P}((Event\_predicate\_symbols \times Terms) \times (BeliefFormula(Belief\_predicate\_symbols \times Terms) \times UserP)).$$

This means that *plan_library* is a *set of pairs*, $\big((e,t),(\varphi,P)\big)$, where (e,t) is a event predicate symbol with its term, and (φ,P) is a belief formula together with a plan body. The *action_description_library* has a similar structure.

We highlight that this axiomatic definition gives the schema for a potentially infinite set of concrete plan and action instances, especially when some components in a plan and an action, such as terms, are left free. To reason about any concrete instances, we can instantiate this schema using set comprehension, which enumerates all of the grounded plans that satisfy the constraints, such as a *typing condition*. For example, consider a BDI plan:

$$e(\mathbf{t}) : \varphi(\mathbf{x_t}, \mathbf{y}) \leftarrow P(\mathbf{x_t}, \mathbf{y}).$$

This plan defines many concrete plans, depending on the values of the variables $\mathbf{t}, \mathbf{x_t}$, and $\mathbf{y}$. To model concrete instances, we apply the constraints to this general plan rule using set comprehension in Event-B shown as follows:

$$\{\mathbf{t}, \mathbf{x_t}, \mathbf{y} \cdot C(\mathbf{t}, \mathbf{x_t}, \mathbf{y}) \mid e(\mathbf{t}) : \varphi(\mathbf{x_t}, \mathbf{y}) \leftarrow P(\mathbf{x_t}, \mathbf{y})\}.$$

This set comprehension expression captures the set of all values of $e(\mathbf{t}) : \varphi(\mathbf{x_t}, \mathbf{y}) \leftarrow P(\mathbf{x_t}, \mathbf{y})$ for all variables of $\mathbf{t}, \mathbf{x_t}, \mathbf{y}$ where the predicate $C(\mathbf{t}, \mathbf{x_t}, \mathbf{y})$ acts as a *constraint* or *typing condition* on the variables. This representation allows us to use a type schema to effectively declare the plan schema and define its instances.

We now show how to encode the plan library and action description library in Fig. 1 for the running example in the rover navigation scenario. Figure 13 gives the context for the encoded plan library and action description library. It extends the base contexts of terms and predicate symbols in Fig. 4 and Fig. 6 (line 2). The constants (line 3) encompass all 6 plans and 5 actions. As a result, the plan library is constructed by unifying all possible plans, in this case, from *plan1* to *plan6*, given by **axm1** (line 6). Each plan (with free variables) represents a set of all possible instantiated plans and is encoded through set comprehension. Here, we only describe the plan (lines 4–5) that is shown in Fig. 2b as follows:

```
start(mission): at(X) <-goal(at(goal_loc), navigate(goal_loc),
                    power(low) ∧ ¬at(goal_loc))
```

We specify this in **axm3** (lines 8–22). Essentially, it encodes a set of plans with three components, namely *triggering_event*, *context*, and *plan_body* (line 8) in Fig. 13. To specify what each of these components is, a list of constraints in the form of logical conjunction is specified (lines 10–20). For example, the constraint $triggering\_event = start \mapsto mission$ (line 11) defines what the triggering event actually is. The context is given using the constructor, $Conjunctive\_formula$, to encode two negated formula (lines 14–16). To specify the *plan_body*, we provide three extra parameters, namely *success_condition*, $failure\_condition$, and *procedural_program* (lines 8–9). Using these, we compose the *plan_body* as a declarative goal on lines 19–20. The set comprehension result (line 21) returns the corresponding plan mapping $triggering\_event \mapsto (context \mapsto plan\_body)$ for any values satisfying the constraints, yielding the set representation of *plan*1. Due to space limits, we omit the encoding for the remaining plans and actions.

3.6 Faithful Encoding

Our encoding of the CAN syntax into Event-B is *faithful* in the following sense: (i) *coverage*—every well-formed CAN object in the syntactic categories that we model (terms, predicate symbols, belief formulas, agent programs, libraries) has a corresponding Event-B representation; and (ii) *uniqueness*—the encoding is *injective*, i.e. distinct CAN objects do not collapse to the same Event-B object.

Coverage follows because the encoding mirrors the CAN grammar: base alphabets (terms and predicate symbols) are introduced axiomatically, and the inductive fragments (e.g. belief formulas and agent programs) are defined using Event-B inductive data types by recursively translating base alphabets and then rebuilding the same syntactic representation in Event-B.

Uniqueness holds because (1) different CAN syntax elements are represented using distinct constructors, and (2) data type constructors in Event-B are injective in their arguments. The equality of encoded objects in Event-B implies the equality of the corresponding sub-components in CAN. For collections (e.g. belief bases), injectivity lifts elementwise. As a result, the encoding is a lossless syntactic encoding of CAN into Event-B.

4 Belief-Action Reasoning for CAN in Event-B

Our focus in this paper is the faithful encoding of CAN *syntax* into Event-B theories and contexts. Full reasoning about BDI behaviour typically requires an operational semantics (e.g. plan selection and event handling), which is our next step. Still, even without that semantics, we can already support useful agent reasoning via two standalone components: the *belief base* and the *action description library*. Concretely, we can (i) evaluate action applicability against the current belief base, and (ii) execute actions to check belief-based safety invariants.

```
MACHINE GenericActionExecution
SEES InitialBeliefBaseExternalEventsRoboticCleaning,
     PlanLibraryActionDescriptionLibraryRoboticCleaning
VARIABLES belief_base
INVARIANTS
  inv1: belief_base ∈ ℙ(Belief_predicate_symbols × Terms)
EVENTS
  INITIALISATION
  THEN
    belief_base := Initial_belief_base
  END
  ExecuteAction
  ANY a, pre, adds, dels
  WHERE
    grd1: a ↦ (pre ↦ (adds ↦ dels)) ∈ action_description_library
    grd2: belief_entail(belief_base, pre)
  THEN
    act1: belief_base := (belief_base \ dels) ∪ adds
  END
```

Fig. 15. Generic action machine for $\langle \mathcal{B}, \Lambda \rangle$.

4.1 Belief Entailment

BDI languages assume a belief entailment mechanism, i.e. whether a belief formula φ follows from a belief base, $\mathcal{B}$. Here, $\mathcal{B}$ is a set of grounded belief atoms, and belief formulas are given by the inductive data type in Fig. 10. We encode belief entailment via a predicate operator **belief_entail** (Fig. 14), defined by recursion over φ: under a closed-world assumption, an atom holds iff it is in $\mathcal{B}$; conjunction/disjunction follow recursively; negation is logical negation; and $\top/\bot$ are tautology/contradiction. This suffices for evaluating action pre-conditions against the current belief base without a full BDI deliberation cycle.

```
MACHINE ExplicitActionExecution
REFINES GenericActionExecution
...
EVENTS
  Charge
  REFINES ExecuteAction
  ANY c
  WHERE
    grd1: c ∈ {init_loc_charge, WP1_charge, WP2_charge, goal_loc_charge}
    grd2: at ↦ c ∈ belief_base
    grd3: power ↦ low ∈ belief_base
  WITH
    act:  act = charge ↦ c
    pre:  pre = Conjunctive_formula(Belief_atom(at ↦ c), Belief_atom(power ↦ low))
    adds: adds = {power ↦ high}
    dels: dels = {power ↦ low}
  THEN
    act1: belief_base := (belief_base \ {power ↦ low}) ∪ {power ↦ high}
  END
```

Fig. 16. Explicit action execution machine for $\langle \mathcal{B}, \Lambda \rangle$

4.2 A Belief-Based State Machine

We now present an Event-B machine that connects the encoded CAN syntax to a generic belief-based transition machine. This machine maintains an agent configuration $\langle \mathcal{B}, \Lambda \rangle$: the belief base $\mathcal{B}$, and a set of actions Λ. The following Event-B machine in Fig. 15 captures this belief-based transition system through executing actions. **INITIALISATION** event initialises the belief base (lines 8–11). **ExecuteAction** event (lines 12–29) non-deterministically selects an action, a, from the action description library where $belief\_entail(belief\_base, pre)$ ensures that the action is applicable only when its pre-condition holds (which also eliminates all actions with free variables) in the current belief base. The effects of an action apply standard add/delete effects, yielding a transition system over belief states, and is sufficient to support invariant-based reasoning about safety properties that are expressible purely in terms of beliefs.

```
1  MACHINE PlanActionExecution
2  REFINES ExplicitActionExecution
3  INVARIANTS
4     inv2: power ↦ empty ∉ belief_base
5  EVENTS
6     Charge_plan
7     REFINES Charge
8     ANY c
9     WHERE
10       grd1: needToCharge = TRUE
11       grd2: at ↦ GoToCharge ∈ belief_base
12       grd3: power ↦ low ∈ belief_base
13    WITH
14       c:   c = GoToCharge
15    THEN
16       act1: belief_base := (belief_base \ {power ↦ low}) ∪ {power ↦ high}
17       act2: needToGobackFromCharge := TRUE
18       act3: needToCharge := FALSE
19    END
```

Fig. 17. Plan-driven action execution machine for $\langle \mathcal{B}, \Lambda \rangle$..

Figure 15 defines a *generic* belief-based belief transition system: each step executes an enabled action from the action description library, and updates the belief base according to the effects of actions. While this machine is convenient, **ExecuteAction** is not explicit in regard to *which* concrete action instance was executed in a given step, which can make the subsequent proving difficult. To expose the concrete action being taken, we refine Fig. 15 to Fig. 16 by refining **ExecuteAction** by action-specific events (e.g. **Charge**), each corresponding to one action in the action description library. Technically, each concrete action-specific event *refines* **ExecuteAction** by (i) fixing the chosen action symbol and its parameters (via witnesses), and (ii) keeping the same belief update. As a result, we can reason about each individual action explicitly in Event-B.

Finally, to capture the user-intent principles [25] imposed by a plan library. Following the standard Event-B methodology [4], we refine Fig. 16 to Fig. 17 by adding a small set of *control tokens* that restricts which action may execute next. For brevity, we show only one representative refined event in Fig. 17.

The **Charge_plan** event refines the **Charge** event by additionally requiring the corresponding control flag (`needToCharge=TRUE`) and by using the plan-selected parameter (`GoToCharge`) as the charging location. The belief update remains identical to the underlying action semantics, while the control layer records progress by switching `needToCharge` off and enabling `needToGoback FromCharge`.

We now illustrate how the invariant-based safety reasoning can be performed in our framework. For example, The safety invariant **inv2** $power \mapsto empty \notin belief\_base$ in Fig. 17 does *not* hold for the current plan library, and the counter-example can be traced to the plan `navigate(Z): at(X)` $\wedge$ `next(X,Y)` $\wedge \neg$ `power(empty)` $\leftarrow$ `movement(X,Y); navigate(Z)`. The guard $\neg power(empty)$ is too weak: it still permits `movement(X,Y)` when the battery is *low*, resulting in $power \mapsto empty$. If the guard were strengthened to $\neg power(low)$ (i.e. movement is allowed only when power is *high*), then re-encoding the intent of this revised plan library yields a model in which **inv2** holds, with the proof successfully discharged. This final refined Rodin development, together with its proofs, is publicly available online[1]. We close this section by noting that **inv2** does not distinguish the rover location and therefore excludes even states in which the rover reaches the goal with an empty battery, more refined safety invariants, for example, involving both battery level and location, can likewise be formulated and proved in the same framework as the future work but out of the scope.

5 Related Work

Formal verification of BDI is well-summarised in [26]. We focus on the *modelling artefact* produced by related approaches, and contrast it with ours. The work in [6] translates a restricted finite-state fragment of AgentSpeak into Promela for analysis in Spin model checker. Agent syntax (e.g. predicate/action symbols) is encoded as integers, while core agent structures such as the belief base and events are represented using bounded Promela data structures. The resulting model is essentially an *interpreter-level* Promela program simulating the AgentSpeak.

Related rewriting-based approaches include [19], which translate BDI agent programs into a Maude rewrite theory where a *single algebraic term* encodes the whole agent configuration via nested subterms (e.g. beliefs and events), and reasoning-cycle steps are rewrite rules over this term. Meanwhile, the work in [5] encodes a propositional fragment of CAN as a bigraphical reactive system, where the agent configuration is a bigraph (place graph for structure, link graph for relations) and the operational semantics is given by bigraph reaction rules.

Recent proof-based approaches mechanise BDI agents in higher-order logic using general-purpose proof assistants (notably Isabelle/HOL). The work in [24] provides a deep embedding of a propositional fragment of GOAL: the syntax of formulas and agent constructs are represented as algebraic data types in HOL, and the operational semantics is defined in HOL as transition relations and trace-based satisfaction. The work in [31] models SimpleBDI using Z-Machines that are

[1] https://github.com/Mengwei-Xu/ABZ2026-Rodin-Artefact.

embedded in Isabelle/HOL, yielding a typed state record (beliefs/goals/plans) with pre/post style state-transforming operations for reasoning-cycle steps.

In contrast to the above approaches, our Event-B encoding ensures that the artefacts of the BDI syntax are reusable and modular as a supporting theory. This approach avoids interpreter-style encodings (Promela/Maude/Bigraph) and prover-internal semantics (HOL embeddings), while preserving state-based modelling and tool-supported proof/refinement. Additionally, it enables the support of more expressive plans, including sub-events and concurrency, extending beyond the simplified BDI agents typically used in proof-assistant encodings.

6 Conclusion and Future Work

This paper presented a faithful and reusable encoding of CAN syntax in Event-B by exploiting algebraic modelling in Event-B theories. The key benefit is practical: it replaces the ad-hoc, one-off syntactic encodings that typically underpin BDI verification with a typed and modular theory suite that can be reused across developments and extended systematically, without reducing BDI agents to restricted language fragments.

Importantly, our encoding of CAN syntax already supports proof-based analysis in Rodin without committing to the full CAN semantics: theory-level operators enable reasoning over the truth of beliefs (e.g. entailment), and actions can be linked to an Event-B transition model to support invariant checking over action executions. The next step is to encode the operational semantics of CAN e.g. over sequencing and concurrent agent programs, and declarative goals, enabling full semantic reasoning and proof-based analysis of CAN within Rodin.

References

1. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transfer **12**(6), 447–466 (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Leuschel, M., Schmalz, M., Voisin, L.: Proposals for mathematical extensions for Event-B (2009)
4. Ait-Ameur, Y., Baron, M., Kamel, N., Mota, J.M.: Encoding a process algebra using the Event B method: application to the validation of human-computer interactions. Int. J. Softw. Tools Technol. Transfer **11**(3), 239–253 (2009)
5. Archibald, B., Calder, M., Sevegnani, M., Xu, M.: Modelling and verifying BDI agents with Bigraphs. Sci. Comput. Program. **215**, 102760 (2022)
6. Bordini, R.H., Fisher, M., Pardavila, C., Wooldridge, M.: Model checking Agentspeak. In: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, pp. 409–416 (2003)
7. Bordini, R., Hübner, J., Wooldridge, M.: Programming multi-agent systems in Agentspeak using Jason, vol. 8. John Wiley & Sons (2007)

8. Bourbouh, H., Farrell, M., Mavridou, A., Sljivo, I.: Integration and evaluation of the AdvoCATE, FRET, CoCoSim, and Event-B tools on the inspection rover case study. Tech. rep. (2020)
9. Bourbouh, H., et al.: Integrating formal verification and assurance: an inspection rover case study. In: NASA Formal Methods Symposium, pp. 53–71. Springer (2021)
10. Bratman, M.: Intention, plans, and practical reason (1987)
11. Butler, M., Maamria, I.: Practical theory extension in Event-B. In: Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday, pp. 67–81. Springer (2013)
12. Cardoso, R.C., Farrell, M., Luckcuck, M., Ferrando, A., Fisher, M.: Heterogeneous verification of an autonomous curiosity rover. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 353–360. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_20
13. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: LASER Summer School on Software Engineering, pp. 1–30. Springer (2011)
14. Clavel, M., et al.: Maude manual (version 3.1). SRI International (2020)
15. De Silva, L., Meneguzzi, F.R., Logan, B.: BDI agent architectures: a survey. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI), 2020, Japão. (2020)
16. Dennis, L.A., Farwer, B.: Gwendolen: a BDI language for verifiable agents. In: Proceedings of the AISB 2008 Symposium on Logic and the Simulation of Interaction and Reasoning, Society for the Study of Artificial Intelligence and Simulation of Behaviour, pp. 16–23 (2008)
17. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Autom. Softw. Eng. **19**, 5–63 (2012)
18. Dennis, L.A., Oren, N.: Explaining BDI agent behaviour through dialogue. Auton. Agent. Multi-Agent Syst. **36**(2), 29 (2022)
19. Doan, T.T., Yao, Y., Alechina, N., Logan, B.: Verifying heterogeneous multi-agent programs. In: Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems, pp. 149–156 (2014)
20. Havelund, K., Pressburger, T.: Model checking Java programs using Java pathfinder. Int. J. Softw. Tools Technol. Transfer **2**(4), 366–381 (2000)
21. Hindriks, K.V., De Boer, F.S., Van Der Hoek, W., Meyer, J.J.C.: Agent programming with declarative goals. In: International Workshop on Agent Theories, Architectures, and Languages, pp. 228–243. Springer (2000)
22. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. **23**(5), 279–295 (1997)
23. Holzmann, G.J., Lieberman, W.S.: Design and validation of computer protocols, vol. 512. Prentice hall Englewood Cliffs (1991)
24. Jensen, A.B.: Machine-checked verification of cognitive agents. In: ICAART (1), pp. 245–256 (2022)
25. Kambhampati, S., Mali, A., Srivastava, B.: Hybrid planning for partially hierarchical domains. In: AAAI/IAAI, pp. 882–888 (1998)
26. Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: a survey. ACM Comput. Surv. (CSUR) **52**(5), 1–41 (2019)
27. Milner, R.: The space and motion of communicating agents. Cambridge University Press (2009)

28. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer (2002)
29. Sardina, S., Padgham, L.: A BDI agent programming language with failure handling, declarative goals, and planning. Auton. Agent. Multi-Agent Syst. **23**(1), 18–70 (2011)
30. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative and procedural goals in intelligent agent systems. In: Proc. of KR'02. Morgan Kaufman (2002)
31. Wright, T., Dennis, L.A., Woodcock, J., Foster, S.: FormalVerification of BDI agents. In: The Combined Power of Research, Education, and Dissemination: Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday, pp. 302–326. Springer (2024)