ECAI 2024
U. Endriss et al. (Eds.)

© 2024 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms
of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA240636

A Practical Operational Semantics for Classical Planning in BDI Agents

Mengwei Xu[a,*], **Tom Lumley**[a], **Ramon Fraga Pereira**[b] **and Felipe Meneguzzi**[c]

[a]Newcastle University, U.K.
[b]The University of Manchester, U.K.
[c]The University of Aberdeen, U.K.

Abstract. Implementations of the *Belief-Desire-Intention* (BDI) architecture have a long tradition in the development of autonomous agent systems. However, most practical implementations of the BDI framework rely on a pre-defined plan library for decision-making, which places a significant burden on programmers, and still yields systems that may be brittle, struggling to achieve their goals in dynamic environments. This paper overcomes this limitation by introducing an operational semantics for BDI systems that rely on *Classical Planning* at run time to both cope with failures that were unforeseeable and synthesise new plans that were unspecified at design time. This semantics places particular emphasis on the interaction of the reasoning cycle and an underlying planning algorithm. We empirically demonstrate the practical feasibility and generality of such an approach in an implementation of this semantics within two popular BDI platforms together with in-depth computational evaluation.

1 Introduction

The *Belief-Desire-Intention* (BDI) [4] architecture is a popular autonomous agent framework and forms the basis of, among others, AgentSpeak [29], *An Abstract Agents Programming Language* (3APL) [17], *A Practical Agent Programming Language* (2APL) [7], Jason [3], and *Conceptual Agent Notation* (CAN) [31]. In BDI agents, beliefs represent what the agent knows, desires what the agent wants to bring about, and intentions those desires the agent has chosen to act upon. BDI agents are successful in business [2] and healthcare [5].

The process through which BDI agent architectures reason about what actions to take to achieve their desires, known as *means-ends reasoning* [4], usually relies on a library of plans that are pre-programmed at the design time [29]. BDI agents, at the run time, then choose their own suitable way of achieving any given goal depending on the current situation. Such context-dependent reasoning makes BDI agents responsive to the environment due to their efficiency and scalability, well suited in complex application domains [21, 23].

However, simplifying the general agent decision-making to the much simpler plan selection problem causes an impractical problem for BDI agent programmers to obtain a plan library (if there is one) that can cope with every possible eventuality. Often, a plan library which covers every possible eventuality, may be, unfortunately, unavailable, particularly in a highly dynamic environment. Furthermore, even with a comprehensive plan library, the agent can still face situations in which all of its plans fail in a hostile environment. As a result, it limits the broader applicability and autonomy of BDI agents.

Meanwhile, long-term autonomy requires autonomous systems to remain robust in execution failure and stay persistent in seeking all possible ways to succeed in the assigned tasks. In response, we propose empowering the decision-making capability of BDI agents through *Classical Planning*. In contrast to BDI agents, classical planning is an approach where a plan instructing which actions to execute is derived automatically from a model consisting of e.g. the specifications of actions and initial/goal states. Instead of pre-programmed specific procedural knowledge in BDI agents, classical planning is interested in formalising a representation of a planning problem (i.e. model) and finding a general way (i.e. algorithm) to e.g. synthesise a sequence of actions which can start from the initial state and end with the goal state. Our approach aims to formally provide BDI agents with the capacity to use classical planning to generate new plans to address gaps in the plan library and persistent failures.

The BDI community has welcomed integrating classical planning with BDI agents, leading to fruitful contributions as reviewed in [26]. However, current approaches either lack formal semantics (e.g. [24, 25]) or are mainly theoretical without practical implementations (e.g. [9, 34]). This paper introduces new operational semantics to formally incorporate classical planning into BDI agents, specifying when and how the planning process should/can be called, and articulating how a BDI agent executes new plans from a planner. We validate our approach on mainstream BDI software e.g. Jason [3] (an extended version of AgentSpeak) and MCAPL [10] (a verifiability-focused interpreter for BDI agents), demonstrating the practicality and providing computational insights of the overhead of such an integration. We also provide a property of our new semantics that there is no unintended halt due to oversight in semantics rule overage.

Our key contributions are fourfold. First, we develop in Section 3 rich and detailed semantics of the appropriate operational behaviour when classical planning is pursued, succeeded or failed, suspended, or resumed. This semantics specifies when and how classical planning can be called to achieve a goal for which either no pre-defined plan worked or exists, and how it interacts with BDI reasoning. Second, we mathematically prove, through a new semantics property in Section 4, that the integration of classical planning does not indefinitely stall the agent's reasoning, except when the agent completes the goal either successfully or unsuccessfully. Third, we show

* Corresponding Author. Email: mengwei.xu@newcastle.ac.uk

in that BDI agent systems and classical planning are theoretically compatible for principled integration and how to construct a planning problem at run time. Finally, we implement our novel semantics in widely-used BDI software (Section 5), showcasing its applicability, and empirically assess the computational overhead of integration between the agents and a planner in Section 6.

We structure the paper as follows. In Section 2, we recall essential background. In Sections 3, 4 and 5, we develop our approach, prove its soundness, and present an implementation of our approach. Section 6 presents the empirical analysis of our implementation. In Sections 7 and 8, we discuss related work and conclude the paper.

2 Theoretical Background

2.1 BDI Agents

CAN features a high-level agent programming language that captures the essence of BDI concepts without describing implementation details such as data structures. This section briefly summarises the CAN syntax and semantics from Winikoff et al. [33], Sardina and Padgham [30], to which we refer for a more detailed account. 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 approach would apply to other BDI programming languages, confirmed by our practical implementation in Section 5.

2.1.1 Syntax

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. And 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. $\mathcal{B} \setminus \{b\}$). We denote the set of all possible belief atoms—for a *specific program*—as $\overline{\mathcal{B}}$ and any given current belief base is given as $\mathcal{B} \subseteq \overline{\mathcal{B}}$. A *plan library* Π contains the procedures of an agent. It consists of a finite collection of plan-rules 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 corresponds to goals, specifying why the plan is triggered, and the context φ determines *when* the plan-body P is able to handle the given event. For convenience, we call the set of events from the external environment i.e. the external event set, denoted E^e. Finally, the remaining events (which occur as part of the plan body) are called sub-events or internal events.

The plan-body P in a plan-rule $Pl = e : \varphi \leftarrow P$ 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-rule (whose triggering event is such a sub-event) is selected if its context is true and its plan-body is added as an intention in place of this event. Actions act take the form $act = \psi \leftarrow \langle \phi^-, \phi^+ \rangle$, where ψ is the pre-condition, and ϕ^- and ϕ^+ are the deletion and addition 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. We denote the set of actions available to an agent as Λ. 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 [30] for details). In essence, the declarative goal is a procedural task with success/failure guards. 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).

2.1.2 Semantics

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

Fig. 1 summarises rules for evolving any single intention. For example, the rule act handles the execution of an action, when the precondition ψ is met, resulting in a belief state update. Rule $event$ replaces an event with the set of relevant plans, while rule $select$ chooses an applicable plan from a set of relevant plans while retaining un-selected plans as backups. With these backup plans, the rules for failure recovery $\rhd_;$, $\rhd_\top$, and $\rhd_\bot$ enable new plans to be selected if the current plan fails (e.g. due to environment changes). Rules ; and $;_\top$ allow executing plan-bodies in sequence, while rules $\|_1$, $\|_2$, and $\|_\top$ specify how to execute (interleaved) concurrent programs. Rules G_s and G_f deal with declarative goals when either the success condition φ_s or the failure condition φ_f become true. Rule G_{init} initialises persistence by setting the program in the declarative goal to be $P \rhd P$, i.e. if P fails try P again, and rule $G_;$ takes care of performing a single step on an already initialised program. Finally, the derivation rule $G_\rhd$ re-starts the original program if the current program has finished or got blocked (when neither φ_s nor φ_f is true).

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

2.2 Classical Planning

Classical Planning is a foundational approach within the broader field of *Automated Planning*. In classical planning, the task is to generate a sequence of deterministic actions that, when executed in sequence from an initial state, generate a final state that satisfies a goal condition. A classical planning problem is defined as a 5-tuple $\mathcal{C} = \langle S, s_0, S_G, A, f(a, s) \rangle$, where S is a finite and discrete set of states, $s_0 \in S$ is the known initial state, $S_G \subseteq S$ is the non-empty set of goal states, A is the set of actions, and $f(a, s)$ is a deterministic transition function, where $s' = f(a, s)$ is the state that follows s after applying action $a \in A$. A solution to this planning problem, denoted $\pi = sol(\mathcal{C})$, is a sequence of actions that generates a state sequence $s_0, s_1, \cdots, s_{n+1}$ where $s_{n+1} \in S_G$.

We also distinguish between online planning and offline planning. In offline planning, a complete plan—a sequence of actions—is generated before any action is executed. Formally, $\pi = sol_{\text{offline}}(\mathcal{C})$, is a sequence of actions $\pi = a_1, a_2, \ldots, a_n$ that generates a state sequence $s_0, s_1, \ldots, s_{n+1}$ where $s_{n+1} \in S_G$. However, offline plan-

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

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

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

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

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

Figure 1: Intention-level CAN semantics.

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

Figure 2: Agent-level CAN semantics.

ning often does not consider changes that might occur during execution. Online planning, in contrast, generates and executes actions one at a time, responding to the current state, which may change due to external factors or the unexpected outcomes of some previous actions. Formally, $\pi = sol_{\text{online}}(\mathcal{C}) = a$. After executing the action, the cycle "plan one action-execute one action" repeats, adapting to the current state and external changes until the goal state is reached.

3 Semantic Integration

We now discuss how CAN and *Classical Planning* can be integrated into a single framework. To avoid confusion, we use classical planning and planning interchangeably from now on. The resulting framework enables agents to perform planning to provide new behaviours at run time whenever suitable. We start by introducing the concept of declarative intention in the next section.

3.1 Declarative Intentions

In CAN, the intention set Γ is limited to procedural goals. Though the advanced feature of CAN has augmented a procedural goal with declarative guards e.g. in $goal(\varphi_s, P, \varphi_f)$, it still relies on procedural goals that only describe *how* to achieve a given task in a pre-defined manner. Meanwhile, the goal of classical planning is to achieve a declarative state through synthesising actions for execution. As such, we need to answer the question as to which declarative states the classical planning should try to achieve for the BDI agent.

To do so, we modify the intention to be a pair of sets, such that $\Gamma = \langle \Gamma_{pr}, \Gamma_{de} \rangle$ with Γ_{pr} and Γ_{de} a set of procedural and declarative intentions, respectively. It allows us to keep track of both procedural tasks (executed by the BDI engine) and declarative states that tell us *what* the agent wants to achieve (used by classical planning). The set of declarative intentions is furthermore partitioned into the subset of active intentions Γ_{de}^+, and the suspended intentions Γ_{de}^-. As a (slight) abuse of notation, we assume adding an element to Γ_{de}^+ ensures the element is removed from Γ_{de}^- and vice versa. We now define what a declarative intention is, and when and how it can be generated for planning and added to declarative intention set for tracking.

Definition 1. *A declarative intention in a* CAN *agent is of the form* $goal(\varphi_s, \varphi_f)$ *such that* $goal(\varphi_s, \varphi_f) \in \Gamma_{de}$.

Example 1. *Autonomous robots are deployed for earthquake rescue. One of the robots, upon encountering a collapsed bridge, can have a declarative intention* goal(unblocked(path), low(battery)) *which wants to have an unblocked path to allow the robot to access injured individuals and halt the planning when the battery is low.*

This new declarative intention encodes the minimum information of what the planning needs to achieve (i.e., successful state φ_s) and when it is sensible to halt the planning (i.e. failure state φ_f). It is read as "*achieve* φ_s; *failing if* φ_f *becomes true*" and defined to be the element of the declarative intention set Γ_{de}. By convention, φ_s and φ_f are mutually exclusive. We note that though $goal(\varphi_s, \varphi_f)$ is very similar to the construct $goal(\varphi_s, P, \varphi_f)$ in the standard CAN syntax, their nature is very different. Whereas one is to provide a goal state for the planning to generate a new plan at the run time, the other is to execute a pre-defined procedural task with declarative guards specifying the success/failure condition to stop. In the next section, we formalise the new semantics on specifying the situations in which a CAN agent can automatically generate declarative intentions to plan for suiting its needs.

3.2 Generation of Declarative Intentions

The first and most important situation in which declarative intentions are added to the declarative intention set is to recover the unexpected failure of the procedural intention, which its original failure handling cannot help either. For example, the agent has already tried all other plans, but the last one is still blocked. In another word, the planning can be regarded as the last resort before the agent concluded the mission failed. As such, we can empower the agent to be more robust.

Before introducing the new semantics, for notational convenience, we use $plan(goal(\varphi_s, \varphi_f))$ to symbolise the actual procedural intention (executed by the agent) on calling the planning for the related declarative intention $goal(\varphi_s, \varphi_f)$. We formalise shortly how this corresponds to a concrete planning problem in Section 3.3 and show the practical feasibility of such calling in Section 5. Fig. 3a gives three rules specifying when the failure of procedural intentions results in calling planning. A short commentary is as follows. Firstly, all these semantics rules are at the agent level, i.e. stating what it means to execute a complete agent. For example, the rule $A_{recovery}^{act}$ in Fig. 3a says that if an action is blocked (i.e. $\langle \mathcal{B}, act \rangle \nrightarrow$) and the built-in failure handling is not applicable (i.e. $\langle \mathcal{B}, P_2 \rangle \nrightarrow$), then the planning is called to establish the pre-condition of such blocked action $plan(goal(\psi, false)$. Since, there is no other information supplied, by default, the semantic gives a condition called $false$

$$\frac{P = act; P_1 \rhd P_2 \in \Gamma \quad \langle \mathcal{B}, act\rangle \not\rightarrow \quad \langle \mathcal{B}, P_2\rangle \not\rightarrow \quad act : \psi \leftarrow \langle \phi^-, \phi^+\rangle \quad goal(\psi, false) \notin \Gamma_{de}}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E^e, \mathcal{B}, \langle \Gamma_{pr} \setminus \{P\} \cup plan(goal(\psi, false)); act; P_1 \rhd P_2, \Gamma_{de}^+ \cup \{goal(\psi, false)\}\rangle\rangle} \; A_{recovery}^{act}$$

$$\frac{P = e : (\mid \Delta \mid); P_1 \rhd P_2 \in \Gamma \quad \langle \mathcal{B}, e : (\mid \Delta \mid)\rangle \not\rightarrow \quad \langle \mathcal{B}, P_2\rangle \not\rightarrow \quad \varphi \leftarrow P' \in \Delta \quad goal(\varphi, false) \notin \Gamma_{de}}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E^e, \mathcal{B}, \langle \Gamma_{pr} \setminus \{P\} \cup plan(goal(\varphi, false)); e : (\mid \Delta \mid); P_1 \rhd P_2, \Gamma_{de}^+ \cup \{goal(\varphi, false)\}\rangle\rangle} \; A_{recovery}^{internal_event}$$

$$\frac{P = e : (\mid \Delta \mid) \in \Gamma \quad \langle \mathcal{B}, e : (\mid \Delta \mid)\rangle \not\rightarrow \quad \varphi \leftarrow P' \in \Delta \quad goal(\varphi, false) \notin \Gamma_{de}}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E^e, \mathcal{B}, \langle \Gamma_{pr} \setminus \{P\} \cup plan(goal(\varphi, false)); e : (\mid \Delta \mid); P_1 \rhd P_2, \Gamma_{de}^+ \cup \{goal(\varphi, false)\}\rangle\rangle} \; A_{recovery}^{external_event}$$

(a) semantics of adding declarative intentions in CAN semantics: $A_{recovery}^{act}$ for the pre-condition of action not holding, $A_{recovery}^{internal_event}$ for no applicable plan available for an internal event, $A_{recovery}^{external_event}$ for no applicable plan for an external event.

$$\frac{P = goal(\varphi_s, \varphi_f); P_1 \rhd P_2 \in \Gamma \qquad goal(\varphi_s, \varphi_f) \notin \Gamma_{de}}{\langle E^e, \mathcal{B}, \Gamma\rangle \Rightarrow \langle E^e, \mathcal{B}, \langle \Gamma_{pr} \setminus \{P\} \cup plan(goal(\varphi_s, \varphi_f)); P_1 \rhd P_2, \Gamma_{de}^+ \cup \{goal(\varphi_s, \varphi_f)\}\rangle\rangle} \; A_{plan}^{direct}$$

(b) semantics of directly adding declarative intentions for planning through the rule A_{plan}^{direct} in CAN semantics.

Figure 3: New semantics of adding declarative intentions for planning.

that is never true. However, we note, in principle, the actual implementation can override this and supply some more meaningful failure conditions (e.g. time-out). Meanwhile, the declarative intention $goal(\psi, false)$ is also added in Γ_{de} to track it.

The second approach is to allow the agent programmers to access the planning capacity directly. This could be the case where classical planning is the best approach to generate a plan instead of a human counterpart, but it still gives the agent programmers control over where and how the planning should be used. To do so, we abuse the notation to allow the agent programmers to write a declarative intention in the plan body. Fig. 3b says that addressing such a declarative intention is to trigger a planning call to achieve this declarative intention and add it to the declarative intention set as a result.

3.3 Planning Problems of Declarative Intentions

We now examine how the planning calling $P = plan(goal(\varphi_s, \varphi_f))$ corresponds to the formation of a classic planning problem $\mathcal{C}$.

Definition 2. *Given the set of all possible belief atoms $\overline{\mathcal{B}}$, the current belief base $\mathcal{B} \subseteq \overline{\mathcal{B}}$, and the set of actions Λ, we have that $plan(goal(\varphi_s, \varphi_f))$ corresponds to the following Classical Planning problem $\mathcal{C} = \langle S, s_0, S_G, A, f(a, s)\rangle$ such that*

- $S = 2^{\overline{\mathcal{B}}}$
- $s_0 = \mathcal{B}$
- $S_G = \{s \mid s \models \varphi_s, s \in S\}$
- $A = \Lambda$
- $f(a, s) = \left\{ \begin{array}{ll} (\mathcal{B} \setminus \phi^-) \cup \phi^+ & \textit{if } a = \psi \leftarrow \langle \phi^-, \phi^+\rangle,\ \mathcal{B} \models \psi \\ \mathcal{B} & \textit{if } a = \psi \leftarrow \langle \phi^-, \phi^+\rangle,\ \mathcal{B} \not\models \psi \end{array} \right\}$

Example 2. *Following the Example 1, the robot can employ classical planning to generate a novel plan to unblock the path. The current belief base of the robot consists of* blocked(path) *and* materials(nearby)*, and the declarative intention of the robot can be* goal(unblocked(path), low(battery))*. The robot also has actions e.g.* gathermaterials*,* constructpath*, and* testpathstability*. For example, the action* gathermaterials *requires the belief atom* materials(nearby) *as a precondition and its post-effect is to add the belief atom* materials(gathered) *and delete the belief atom* materials(nearby) *to construct the path. As such, we can have, for planning, initial state $s_0 = \{$blocked(path), materials(nearby)$\}$, goal states $S_G = \{s \mid s \models$ unblocked(path)$, s \in S\}$, and e.g. $f($gathermaterials$, s) = (s \setminus \{$materials(nearby)$\}) \cup \{$materials(gathered)$\}$ if $s \models$ materials(nearby).*

3.4 BDI Execution of Planning Solutions

We now consider how to execute the solution to a classical planning problem and how it integrates with CAN semantics. From now on, we also distinguish between online planning [22] and offline planning [19] and give different rules for accommodating each style of solution due to their contrasting nature. We first look at the main scenarios when neither the success nor failure condition of a declarative intention holds before a plan is returned by providing the agent-level rules to add newly generated plans into the procedural intention set.

In *offline* planning, a complete sequence of actions $sol(\mathcal{C}) = \pi = a_0, \cdots, a_n$ is generated first. The rule $A_{exe^{offline}}$ in Fig. 4 specifies the adoption of such a complete sequence of actions from *offline* planning by replacing the previous planning call with the solution for it in the procedural intention. Meanwhile, it also suspends the related declarative intention in the declarative intention set.

In *online* planning, a single action is returned based on current belief states instead of generating the whole plan a priori, and executed immediately by the agent. The next action will be generated based on newly reached belief states. The loop of "*plan one action—execute one action*" iterates until the goal is reached by the agent. As such, we have the rules of $A_{exe^{online}}$ and $A_{replan^{online}}$ in Fig. 4 for *online* planning. The rule $A_{exe^{online}}$ is similar to the rule $A_{exe^{offline}}$ for offline planning. However, besides replacing the planning call with the solution, it also remembers to recall the planning (through the rule $A_{replan^{online}}$) to take the new belief into consideration and plan for the next action. These two interleaved planning and execution repeats until the successful state is achieved, if possible.

We now look at what if the success/failure condition holds *before*, *during*, or *after* the plan solution is being generated. It has two implications: one to the actual planning calling of the declarative intention in the procedural intention set, the other to the actual declarative intentions tracked in the declarative intention set. We first look at the former. The following two intention-level rules G_s^{plan} and G_f^{plan} handle the cases where either the success condition φ_s or the failure condition φ_f holds *before* and *during* the plan solution is being generated. For example, the rule G_s^{plan} says that $plan(goal(\varphi_s, \varphi_f))$ is trivially completed. Meanwhile, the rule G_f^{plan} says that, if used for failure recovery, the final attempt of failure recovery through planning is failed. It triggers the standard semantics of CAN to either back-propagate the failure to the higher goal or remove the intention if it is already at the top of the goal. Once the plan solution is generated, the truth of success/failure condition in $goal(\varphi_s, \varphi_f)$ only affects the declarative intention in the declarative intention set.

$$\frac{P_1 = plan(goal(\varphi_s, \varphi_f)) \;\; P_1; P_2 \in \Gamma_{pr} \;\; sol(\mathcal{C}_1) = \pi \;\; \mathcal{B} \nvDash \varphi_s \;\; \mathcal{B} \nvDash \varphi_f}{\langle E^e, \mathcal{B}, \langle \Gamma_{pr}, \Gamma_{de} \rangle\rangle \Rightarrow \langle E^e, \mathcal{B}, \langle (\Gamma_{pr} \setminus \{P_1; P_2\}) \cup \{\pi; P_2\}, \Gamma_{de}^- \cup \{goal(\varphi_s, \varphi_f)\rangle\rangle} \; A_{exe^{offline}}$$

$$\frac{P_1 = plan(goal(\varphi_s, \varphi_f)) \;\; P_1; P_2 \in \Gamma_{pr} \;\; sol(\mathcal{C}_1) = act \;\; \mathcal{B} \nvDash \varphi_s \;\; \mathcal{B} \nvDash \varphi_f}{\langle E^e, \mathcal{B}, , \langle \Gamma_{pr}, \Gamma_{de}^+ \rangle\rangle \Rightarrow \langle E^e, \mathcal{B}, \langle (\Gamma_{pr} \setminus \{P_1; P_2\}) \cup \{act; activate(goal(\varphi_s, \varphi_f)); P_2\}, \Gamma_{de}^- \cup \{goal(\varphi_s, \varphi_f)\}\rangle\rangle} \; A_{exe^{online}}$$

$$\frac{P_1 = activate(goal(\varphi_s, \varphi_f)) \;\; P_1; P_2 \in \Gamma_{pr}}{\langle E^e, \mathcal{B}, , \langle \Gamma_{pr}, \Gamma_{de}^+ \rangle\rangle \Rightarrow \langle E^e, \mathcal{B}, \langle (\Gamma_{pr} \setminus \{P_1; P_2\}) \cup \{plan(goal(\varphi_s, \varphi_f)); P_2\}, \Gamma_{de}^+ \cup \{goal(\varphi_s, \varphi_f)\}\rangle\rangle} \; A_{replan^{online}}$$

Figure 4: Semantic of executing the solution to a planning problem: $A_{exe^{offline}}$ for executing a complete sequence of actions from the offline planning; $A_{exe^{online}}$ and $A_{replan^{online}}$ for executing an action from the online planning and replan for the next action.

$$\frac{\mathcal{B} \vDash \varphi_s}{\langle \mathcal{B}, plan(goal(\varphi_s, \varphi_f))\rangle \longrightarrow \langle \mathcal{B}, nil\rangle} \; G_s^{plan}$$

$$\frac{\mathcal{B} \vDash \varphi_f}{\langle \mathcal{B}, plan(goal(\varphi_s, \varphi_f))\rangle \longrightarrow \langle \mathcal{B}, ?false\rangle} \; G_f^{plan}$$

In addition, a trivial goal can be safely terminated:

$$\frac{}{\langle \mathcal{B}, plan(goal(\top, \bot)))\rangle \Rightarrow \langle \mathcal{B}, nil\rangle} \; P_\top$$

When no solution is found to achieve goal state φ_s, the BDI agent also reports the failure of the planning (i.e. $?false$).

$$\frac{P = plan(goal(\varphi_s, \varphi_f)) \;\; sol(\mathcal{C}) = \emptyset}{\langle \mathcal{B}, P\rangle \to \langle \mathcal{B}, ?false\rangle} \; P_\bot$$

Finally, we look at the second implication of the truth of the success/failure condition by providing the agent-level rules to remove the achieved/failed declarative intentions from the declarative intention set. The rule $A_{s,f}^{plan}$ stops tracking a declarative intention once either the success or failure condition holds. The rule $A_\top^{plan}$ automatically removes the trivial declarative intention from the declarative intention set. The rule $A_\bot^{plan}$ removed the declarative intention for which the planning gave no solution (i.e. by the rule $P_\bot$). The actual blocked procedural task of calling planning $plan(goal(\varphi_s, \varphi_f))$ will remain and be handled by the normal failure handling in CAN.

$$\frac{\mathcal{B} \vDash \varphi_s \vee \varphi_f}{\langle E^e, \mathcal{B}, \langle \Gamma_{pr}, \Gamma_{de}\rangle\rangle \longrightarrow \langle E^e, \mathcal{B}, \langle \Gamma_{pr}, \Gamma_{de} \setminus goal(\varphi_s, \varphi_f)\rangle\rangle} \; A_{s,f}^{plan}$$

$$\frac{goal(\top, \bot) \in \Gamma_{de}}{\langle E^e, \mathcal{B}, \Gamma\rangle \longrightarrow \langle E^e, \mathcal{B}, \langle \Gamma_{pr}, \Gamma_{de} \setminus goal(\top, \bot)\rangle\rangle} \; A_\top^{plan}$$

$$\frac{\langle \mathcal{B}, plan(goal(\varphi_s, \varphi_f))\rangle \nrightarrow}{\langle E^e, \mathcal{B}, \Gamma\rangle \longrightarrow \langle E^e, \mathcal{B}, \langle \Gamma_{pr}, \Gamma_{de} \setminus goal(\varphi_s, \varphi_f)\rangle\rangle} \; A_\bot^{plan}$$

4 Semantics Property

We have introduced new semantics to integrate classical planning into the BDI reasoning cycle. However, flawed semantics could undesirably halt the agent's operation. Our primary focus is to ensure that these newly added behaviours for classical planning do not cause the agent's reasoning to get stuck. To this end, we analyse all possible scenarios whether the planning problem is solved by the planning or not, or whether the success/failure conditions of a declarative intention $goal(\varphi_s, \varphi_f)$ are met before, during, or after generating the plan. Again, we have that φ_s and φ_f are mutually exclusive.

Theorem 1 (Strong Progress)**.** *Our new semantics exhibit the strong progress property such that, for any agent state configuration when calling planning, at least one semantic rule applies to it.*

Proof. We prove this by exhaustively enumerating all scenarios, ensuring that the procedural intention on actual planning calling $plan(goal(\varphi_s, \varphi_f))$ and the declarative intention $goal(\varphi_s, \varphi_f)$ are appropriately addressed where $sol(\mathcal{C})$ denotes the solution of the planning problem $\mathcal{C}$, corresponding to $plan(goal(\varphi_s, \varphi_f))$.

- φ_s holds before or during $sol(\mathcal{C})$ is being generated (either with solution or no solution returned): rule G_s^{plan} and $A_{s,f}^{plan}$ apply.
- φ_s holds after $sol(\mathcal{C})$ is returned:
  1. $sol(\mathcal{C}) = \emptyset$: rule $P_\bot$ and $A_{s,f}^{plan}$ apply
  2. $sol(\mathcal{C}) \neq \emptyset$: one rule from Fig. 3 (depending on the nature of planning calling) and $A_{s,f}^{plan}$ apply.
- φ_s does not hold before, during, or after $sol(\mathcal{C})$ is being generated
  1. $sol(\mathcal{C}) = \emptyset$: rule $P_\bot$ and $A_\bot^{plan}$ apply
  2. $sol(\mathcal{C}) \neq \emptyset$: one rule from Fig. 3 (depending on the nature of planning calling) applies, and no applicable rule for $goal(\varphi_s, \varphi_f)$ by default.

The proof is completed by replacing φ_s with φ_f and e.g. the rule G_s^{plan} with G_f^{plan}, considering they are mutually exclusive. □

5 Practical Implementation

We have given our new semantics and we now show the practicality of our semantics by implementing it across multiple existing BDI platforms, including Jason [3][1] and MCAPL [10][2]. Both platforms are based on the AgentSpeak language, with minor adaptations. These variations do not affect the fundamental BDI reasoning, such as plan selection and action execution, but they do influence our specific implementations. For example, MCAPL has no failure handling, necessitating immediate planning upon failure. This again showcases the broader applicability of our approach to different agent frameworks. Due to space limitations, we omit the full implementation details in MCAPL, which is very similar to Jason, and instead provide a detailed explanation of integrating Jason with classical planning to illustrate our approach.

5.1 Implementation Design

Fig. 5 depicts our implementation design where the key is on when to call a planner (e.g. the tool that employs planning algorithms to generate plans) and what the agent should output as the input of the planner. We choose the Planning Domain Definition Language (PDDL) [14]—the de-facto standard planning language for specifying planning problems for classical planning. As a result, we can use any planner which supports the PDDL language. Each PDDL input for a planner consists of two parts: (1) a domain file containing the set of actions that can be taken and (2) a particular instance of a planning problem, including the initial state and goal state. The top of Fig. 5 shows the correspondence of such translations where the PDDL domain can be generated at the design time (as the set of actions available in BDI agents often remains unchanged), whereas the

[1] https://github.com/Mengwei-Xu/Jason-Automated-Planning
[2] https://github.com/Mengwei-Xu/MCAPL-Automated-Planning

bottom half of Fig. 5 details the implementation of semantics when the PDDL problem file should be generated at run time.

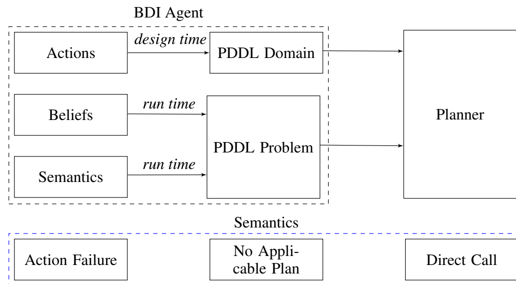

Figure 5: Overview of the implementation design.

5.2 Jason

Jason is one of the most well-known BDI platforms—a fully-fledged interpreter for a much-improved version of AgentSpeak. In our implementation, we have modified Jason's default approach to enable the external call of planning for the following three situations.

5.2.1 Action Failure

In Jason, action specifications are defined within the agent's environment. This is often done using Java methods, for example: if(action.getFunctor().equals("myAction"))effectsAction. We use this default setting but introduce a more structured format for organising agent actions as $act : \psi \leftarrow \langle \phi^-, \phi^+ \rangle$, allowing the automatic translation of BDI actions into actions for classical planning defined in Section 3.3. If an action's precondition is met, the execution proceeds. If not, we initiate a recovery process, beginning with the extraction of the agent's current beliefs and setting the precondition of the blocked actions as the goal state.

5.2.2 No Applicable Plan Failure

To address failures due to no applicable plans available, we must first be able to detect this failure. We achieve this in Jason by adding a goal listener for such goal failures and triggering a recovery process when they occur. During recovery, we retain a copy of the relevant plans related to the unachievable goal. By default, we select the first relevant plan and use its context as the goal state for planning. We note this method of selecting the plan's context could be optimised. For instance, we could improve this by initially conducting a heuristic search to determine the "closest" achievable context before the full planning (which is one of our future works).

5.2.3 Direct Planning Call

Our approach to implementing direct planning calls utilises the straightforward extensibility offered by Jason through user-defined internal actions, which are programmed as functions in Java. In our case, we simply implement such internal action as *.planning(goal_state)* where . denotes an internal action as opposed to normal actions executed by the agent. Once such an internal action is executed, it starts extracting the current belief base, actions, and goal state, and then passes it to the planner for a solution.

5.2.4 Offline/Online Planning

We also support the employment of both online and offline planning. Offline planning is straightforward. After calling the planner, the agent simply executes the sequence of actions returned from the planner. Once all actions have been executed, the agent proceeds to either the blocked program or the next program. Unlike offline planning, the online planner simply returns an action. The agent first executes this action and stops if the goal state is achieved. If not, it repeats the planning call but based on the current belief base until reaching the goal state.

5.3 Planner Consideration

In our current approach, we view the planning as an external entity. As such, we translate the knowledge (e.g. the current beliefs and actions) in BDI agents to initial states and actions in PDDL with three main steps. The first is to translate the belief predicates in the belief base of a BDI agent as the predicate declaration in (: init) in PDDL; The second is to translate the success condition φ_s in $plan(goal(\varphi_s, \varphi_f))$ as the predicate declaration in (: goal) in PDDL. The final one is to translate actions in BDI agents into action declaration (i.e. (: action... : precondition... : effect...)) in PDDL.

For the sake of feasibility, we chose the Fast-Forward planner [18] for the offline setting and PDDL4J [28] (which we modified to output the first best action) for the online setting. In principle, depending on the domain and environment, programmers can select planners that balance optimality and speed. For instance, a planner that quickly generates a satisfactory plan may be preferable to one that creates the best plan but requires more time.

6 Experimental Evaluation

We evaluate the computational overhead of our approach, focusing on the time taken for information exchange between BDI agents and the planner. This evaluation is to show that it is possible, with our semantics, to use the BDI machinery as a practical reasoning mechanism tied to classical planning as a means-end reasoner without incurring an overhead that would be impractical. Specifically, we measure the time from the planning calling to PDDL file generation, and from the complete generation of a plan to its integration into the agent's procedural intentions for execution. The actual time for the planner to generate a plan is planner-specific, and we discuss the pros and cons of this approach in Section 8.

6.1 Evaluation Setting

For generality, our evaluation is based on a set of randomly generated, synthetic parameterised belief atoms (resp. the sequence of actions) in BDI agents (resp. the actual planner). By varying (1) the number of belief atoms that the agent needs to translate into PDDL files and (2) the length of the action sequence that the planner returns, we can evaluate the overhead of our approach. The mean time is calculated after many repetitive runs to get a reliable time measure. All results are obtained on a laptop with the specification of 8-core AMD EPYC 7B13, 2.45 GHz, Ubuntu 22.04.4 LTS (64-bit).

6.2 BDI Agents to Planners

When a planning call occurs, the agent generates a PDDL problem and PDDL domain file. Since the number of actions available to the

agent is unlikely to change on each planning occasion, the PDDL domain file can be translated at the design time (Fig. 5), reducing the run-time overhead. Fig. 6 shows a near-linear correlation between the performance and the total number of belief atoms in the PDDL problem file, including both the initial and goal states. Notably, translation time stays under 35 milliseconds with up to 1000 belief atoms.

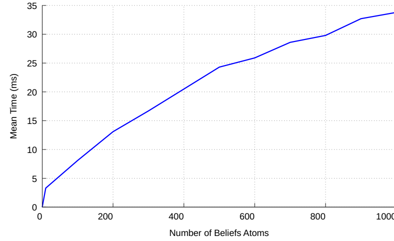

Figure 6: The time of generating PDDL problem file increases near-linearly with the total size of belief atoms in the BDI agent.

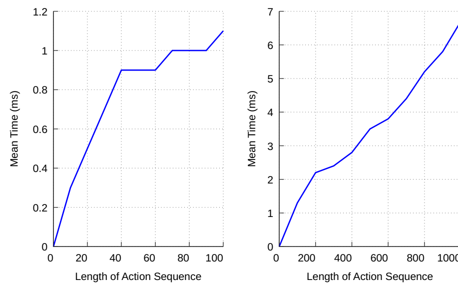

Figure 7: The time to adopt a plan from a planner into the agent's procedural intentions increases near-linearly overall (the right) with the length of the action sequence and varies when under 100 actions.

6.3 *Planners to BDI Agents*

Similarly, we measure the time from the moment when a plan is generated by a planner to the moment the agent has adopted it in the procedural intention ready for execution. Fig. 7 shows that it is even faster than PDDL translation with the near-linear property overall with the size of the sequence of actions (the right) and fluctuating when under 1ms with sequences under 100 actions (the left).

7 Related Work

Work on automated planning within BDI agents integrates one of two key types of planning algorithm: classical, state-space planners in the STRIPS tradition [13]; and hierarchical planning, such as hierarchical task network (HTN) [12]. While there are numerous approaches to the use of automated planning (e.g. Ingrand [20], Despouys and Ingrand [11]), we focus on work that is either recent, or that bears a more direct connection to the approach taken in this paper. For an overview of the long tradition of integrating automated planning in BDI agents, we refer the reader to Meneguzzi and Silva [26].

Recent work on the first type of planning, such as [1], focuses primarily on recovering failed actions by calling a planning algorithm to establish the precondition of a blocked action. Similarly, Stringer et al. [32] employs classical planning to patch the plans that fail as a result of the actions that cease to perform as expected. Though classical planning is crucial for failure recovery in BDI agents, it overlooked the benefits of classical planning, relieving the agent programmer from creating detailed plans. Crucially, similar to earlier work Meneguzzi and Luck [24], these approaches lack an operational semantics. By contrast, most work on planning with BDI focuses on the second type of planning, since it relies on the similarities of HTN and BDI agents [8] and in the work of Sardina et al. [31]. For example, the selection of a plan for an event is similar to the decomposition of an abstract task into a less abstract task. As such, the integration of HTN is proposed to predominantly help BDI agents perform lookahead deliberation over the pre-defined plans.

Conversely, research in the planning community has recently taken an interest in the actor's interaction with planning processes, known as *Automated Planning and Acting* e.g. [16]. This view shows the efficiency of planning in formulating strategies but underscores the complexity of action execution beyond mere plan execution. Hence, an actor with operational models supporting planning thus becomes indispensable for bridging this gap. For example, the work [27] proposed a hierarchical operational model that supports both planning and acting. In particular, it uses the Refinement Acting Engine (RES), a system which is inspired by the Procedural Reasoning System (PRS) [15], and PRS is one of the earliest implementations inspired by the BDI framework. At its core, same as us, it still focuses on an acting perspective (e.g. BDI agents), extended with planning capabilities. However, our approach takes the pragmatic position from the BDI literature that, given the high cost of planning, an agent must selectively deploy it in scenarios where its impact is maximised, as opposed to the consistent utilisation of planning at each decision step for actor in [27]. As such, we can mitigate the extensive programming effort typically required in [27] by comprehensively annotating options with e.g. utility functions (which may be hard to estimate at design time). Furthermore, our approach is not tied to any specific BDI agents, demonstrated in the implementation of some of the most popular BDI agent platforms to highlight its versatile applicability.

8 Discussion and Conclusion

We employ classical planning as an external technique through the standardised PDDL language, making our approach highly accessible to many actual planners the users can choose. Though it means that the plan generation time is not under our control, users can opt for high-performance planners described in [6] for more efficiency.

Our approach also supports both online and offline planning. Unlike the more agreeable choice of selecting highly performance planners, we believe it is a domain-dependent question on when to use online or offline planning. As a general thumb of rule, the more dynamic the environment is, the more likely online planning is a better choice. For instance, if the success state holds without completely executing the full sequence of actions or some of these generated actions are also blocked, then online planning may be a sensible choice.

Currently, planning acts as a fallback when usual agent operations are obstructed e.g. to generate a new plan to establish the required action precondition. If these plans generated by the planner also face obstacles, planning can still be re-invoked for further attempts (though online planning would be recommended here). However, if no plan is found by the planner, the agent acknowledges the failure to e.g. accept an event unaccomplished. This is our current semantic design, though in principle, there are no theoretical/practical barriers to implementing a retry limit for planning before the agent gives up.

Overall, our approach is theoretically appealing with a formal semantic (with strong progress property) and practically feasible, supporting any PDDL planner and both online and offline planning within a wide range of BDI software. It showcases the principle alignment between BDI agents and classical planning, and, importantly the effectiveness of these two for long-term autonomy.

References

[1] H. Baitiche, M. Bouzenada, and D. E. Saidouni. Failure recovery mechanism for BDI agents based on abilities and discovery protocols. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–8, 2024.

[2] S. S. Benfield, J. Hendrickson, and D. Galanti. Making a strong business case for multiagent technology. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent systems (AAMAS)*, 2006.

[3] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.

[4] M. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, 1987.

[5] L. Braubach, A. Pokahr, and W. Lamersdorf. Negotiation-based Patient Scheduling in Hospitals. In *Advanced Intelligent Computational Technologies and Decision Support Systems*, pages 107–121. 2014.

[6] A. B. Corrêa, G. Frances, M. Hecher, D. M. Longo, and J. Seipp. Scorpion Maidu: Width Search in the Scorpion Planning System. In *Proceedings of International Planning Competition (IPC)*, pages 42–55, 2023.

[7] M. Dastani. 2APL: A Practical Agent Programming Language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[8] L. De Silva and L. Padgham. A Comparison of BDI based Real-time Reasoning and HTN based Planning. In *Proceedings of the 17th Australian Joint Conference on Artificial Intelligence*. Springer, 2005.

[9] L. De Silva, S. Sardina, and L. Padgham. First Principles Planning in BDI Systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2009.

[10] L. A. Dennis. The MCAPL Framework including the Agent Infrastructure Layer an Agent Java Pathfinder. *Journal of Open Source Software*, 3(24):617, 2018.

[11] O. Despouys and F. c. o. F. Ingrand. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *Proceedings of the 5th European Conference on Planning*, 2000.

[12] K. Erol, J. Hendler, and D. S. Nau. HTN Planning: Complexity and Expressivity. In *Proceedings of 12th the AAAI Conference on Artificial Intelligence*, 1994.

[13] R. Fikes and N. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3-4): 189–208, 1971.

[14] M. Fox and D. Long. PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[15] M. P. Georgeff and A. L. Lansky. Reactive Reasoning and Planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1987.

[16] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[17] 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*, 2(4):357–401, 1999.

[18] J. Hoffmann. FF: The Fast-Forward Planning System. *AI Magazine*, 22 (3):57–57, 2001.

[19] J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. In *Journal of Artificial Intelligence Research*, pages 253–302, 2001.

[20] F. Ingrand. PROSKILL: A Formal Skill Language for Acting in Robotics. *CoRR*, abs/2403.07770, 2024.

[21] N. R. Jennings and S. Bussmann. Agent-Based Control Systems. *IEEE Control Systems*, 23(3):61–74, 2003.

[22] T. Keller and P. Eyerich. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

[23] S. D. McArthur, E. M. Davidson, V. M. Catterson, A. L. Dimeas, N. D. Hatziargyriou, F. Ponci, and T. Funabashi. Multi-Agent Systems for Power Engineering Applications - Part I: Concepts, Approaches, and Technical Challenges. *IEEE Transactions on Power Systems*, 22(4): 1743–1752, 2007.

[24] F. Meneguzzi and M. Luck. Composing High-Level Plans for Declarative Agent Programming. In *International Workshop on Declarative Agent Languages and Technologies*. Springer, 2007.

[25] F. Meneguzzi and M. Luck. Leveraging New Plans in AgentSpeak (PL). In *International Workshop on Declarative Agent Languages and Technologies*. Springer, 2008.

[26] F. Meneguzzi and L. Silva. Planning in BDI Agents: A Survey of the Integration of Planning Algorithms and Agent Reasoning. *The Knowledge Engineering Review*, 30:1–44, 2015.

[27] S. Patra, J. Mason, M. Ghallab, D. Nau, and P. Traverso. Deliberative Acting, Planning and Learning with Hierarchical Operational Models. *Artificial Intelligence*, 299:103523, 2021.

[28] D. Pellier and H. Fiorino. PDDL4J: A Planning Domain Description Library for Java. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1):143–176, 2018.

[29] 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*, pages 42–55. Springer, 1996.

[30] S. Sardina and L. Padgham. A BDI Agent Programming Language with Failure Handling, Declarative Goals, and Planning. *Autonomous Agents and Multi-Agent Systems*, 23:18–70, 2011.

[31] S. Sardina, L. De Silva, and L. Padgham. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, 2006.

[32] P. Stringer, R. C. Cardoso, C. Dixon, M. Fisher, and L. A. Dennis. Adaptive Cognitive Agents: Updating Action Descriptions and Plans. In *European Conference on Multi-Agent Systems (EMAS)*. Springer, 2023.

[33] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and Procedural Goals in Intelligent Agent Systems. *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, 2002.

[34] M. Xu, K. Bauters, K. McAreavey, and W. Liu. A Formal Approach to Embedding First-Principles Planning in BDI Agent Systems. In *Proceedings of the International Conference on Scalable Uncertainty Management (SUM)*. Springer, 2018.