Generalised BDI Planning

Felipe Meneguzzi
University of Aberdeen & PUCRS
Aberdeen, Scotland, UK
felipe.meneguzzi@abdn.ac.uk

Ramon Fraga Pereira
University of Manchester & UFRGS
Manchester, England, UK
ramon.fragapereira@manchester.ac.uk

Nir Oren
University of Aberdeen
Aberdeen, Scotland, UK
n.oren@abdn.ac.uk

ABSTRACT

Agent interpreters based on the *Beliefs*, *Desires*, and *Intentions* (BDI) model traditionally perform means-ends reasoning using plan libraries composed of reactive planning rules. However, the design of such rules often imposes a heavy knowledge engineering burden on a designer, and trades off flexibility for runtime efficiency. This use of planning rules originates from the limitations of planning technology at the time of the first BDI implementations. While these limitations have gradually been overcome by the integration of various types of planning into existing BDI theories, the corresponding interpreters remain fundamentally plan-library based. In this paper, we develop a novel BDI agent architecture driven by *generalised planning* as means-ends reasoning, in a radical departure from existing architectures. This architecture has two key properties. First, it more closely resembles the foundations of BDI logic and reasoning. Second, it offers substantial gains in efficiency in comparison with an architecture driven by *classical planning*.

KEYWORDS

BDI, Automated Planning, Generalized Planning, Autonomous Agents

ACM Reference Format:
Felipe Meneguzzi, Ramon Fraga Pereira, and Nir Oren. 2025. Generalised BDI Planning. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 9 pages.

1 INTRODUCTION

Agent architectures based on *Beliefs*, *Desires*, and *Intentions* (BDI) [3] have inspired a long tradition of research on pragmatic models of autonomous reasoning. The key insight of such architectures is the separation of theoretical reasoning (i.e., epistemological reasoning, concerned with the truth), and practical reasoning (i.e., means-ends reasoning, concerned about goal-directed action) in resource-bounded agents. This separation allows the agent to deal with perception separately from planning its future-directed actions, and to direct their computational effort towards practical reasoning [25]. A BDI agent limits the computational cost of practical reasoning at any given time by committing to specific goals and courses of action that should be achievable, whilst allowing for such commitments to be revised [4]. Generating such courses of actions, i.e., by performing *automated planning* [17], is a costly process, which led to practical implementations of BDI agents generally avoiding using fully fledged planning algorithms. Instead, most BDI agent implementations rely on a form of reactive planning (via crafted plan libraries) that, while conceptually similar to hierarchical planning [15], requires substantial engineering effort to replicate the properties of BDI agents and logics [8]. While avoiding potentially computationally expensive planning algorithms made sense in the past, developments in automated planning now provide a practical mechanism for means-ends reasoning [18].

Recent research has gradually introduced first principles planning capabilities via automated planning algorithms into BDI agents [12, 38]. However, the resulting architectures inherit two key limitations, previously highlighted by Pereira and Meneguzzi [33] and Reed et al. [37]. First, these approaches remain fundamentally tied to the reactive planning reasoning cycle of traditional BDI architectures [24]. Second, the means-ends reasoning provided by single instances of *classical planning* processes focuses on a single desire, departing from the implicit capability of BDI reasoning to reason *globally* over all of an agent's desires.

We seek to address these limitations through the introduction and development of GePetto, a BDI agent architecture completely driven by *Generalised Planning* [22], which is a type of planning that naturally represents multiple individual goals, thereby aligning more closely to the means-ends reasoning required by BDI agents. Our key contributions are threefold. First, we formalise a fully-fledged BDI agent interpreter that relies on a generalised planner to drive its means-ends reasoning process (Section 3). Second, we show that this interpreter automatically complies with the key properties of BDI logics [3, 8] in ways that reactive planning agents do not (Section 4). Third, we empirically evaluate GePetto across two different scenarios under specific conditions where generalised planning enable GePetto's means-ends reasoning to reduce the number of planning calls (Section 5). This results in a more efficient BDI reasoning process, both in terms of planning time and the number of achieved desires (goals), compared to an agent that relies on classical planning.

2 BACKGROUND AND NOTATION

In this section, we present a comprehensive overview of the foundational concepts and essential notation in *Classical Planning*, *Generalised Planning*, and *BDI Planning*.

2.1 Planning

The *environment* in which our autonomous agents act towards achieving their goals follows the formalism of *Classical Planning* [17]. Here, a *domain model* representing the environment is assumed to be *fully observable*, and *discrete*, and the actions' outcomes (i.e., effects) are *deterministic*. In our experiments the environment is not static, and may change independently of the agent's actions.

A *planning domain model* Ξ is a tuple $\langle \mathcal{F}, \mathcal{A} \rangle$ where $\mathcal{F}$ is a set of *fluents* (i.e., environment properties) and $\mathcal{A}$ is a set of *actions*

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowé (eds.), May 19 – 23, 2025, Detroit, Michigan, USA. © 2025 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

where every action $a \in \mathcal{A}$ has a *positive cost*, denoted as $cost(a)$, and is associated with its own set of *preconditions*, *add* and *delete* lists: $pre(a)$, $add(a)$, $del(a)$. We define a state S as a finite set of positive fluents $f \in \mathcal{F}$ that follows the *closed world assumption* so that if $f \in S$, then f holds, or is true, in S. We also assume a simple inference relation $\models$ such that $S \models f$ iff $f \in S$; and $S \models f_0 \wedge ... \wedge f_n$ iff $\{f_0, ..., f_n\} \subseteq S$. An action $a \in \mathcal{A}$ is *applicable* to a state S (written $applicable(a, S)$) iff $S \models pre(a)$, and when executed generates a new state S' such that $S' \leftarrow (S \cup add(a)) \backslash del(a)$. An action execution function $\gamma(S, a)$ returns S' if $applicable(a, S)$ and $\perp$ otherwise.

A *planning problem* $\mathcal{P}$ is a tuple $\langle \Xi, O, s_0, s_g \rangle$ where: Ξ is a planning domain as described above; O is a finite set of *environment objects*; $s_0 \subseteq \mathcal{F}$ is the *initial state*; and $s_g \subseteq \mathcal{F}$ is a *goal state*. A *solution* to the planning problem $\mathcal{P}$ is a *plan* $\pi = [a_1, ..., a_n]$ that transforms s_0 into a state $S \models s_g$ after executing actions $a_1, \ldots, a_n$ sequentially, resulting in a state where goal s_g holds. We define a plan execution function $\gamma(\Xi, S, \pi)$ that returns the state resulting from the sequential execution of each action in a plan within a domain Ξ. The cost of a plan $\pi = [a_1, ..., a_n]$ is $cost(\pi) = \sum_i cost(a_i)$. A plan π^* is *optimal* if there is no other plan π' that is a solution for $\mathcal{P}$ such that $cost(\pi') < cost(\pi^*)$. The purpose of a planning algorithm is to find a plan π which is a solution to a planning problem $\mathcal{P}$. We call $\textsc{planner}(\Xi, s_0, s_g)$ an algorithm that *solves* $\mathcal{P}$ by providing such a solution plan.

2.2 Generalised Planning

As defined by [7, 22, 43], a *generalised planning problem* $\mathcal{GP}$ is a tuple $\{\mathcal{P}_0, \mathcal{P}_1, ..., \mathcal{P}_N\}$ that represents the task of solving a set of $N \geq 2$ planning problems that share some common structure. A *solution* to a generalised planning problem $\mathcal{GP}$ is a *generalised plan* Π that solves a $\mathcal{GP}$. We follow the formalism of Segovia-Aguas et al. [40] and define *generalised plans* as *planning programs*. A generalised plan Π (alternatively, planning program) has a control flow that encompasses a compact representation that captures different ways to solve a set of possibly distinct planning problems $\{\mathcal{P}_0, \mathcal{P}_1, ..., \mathcal{P}_N\}$. More formally, a generalised plan Π is a planning program that has a sequence of n instructions $\Pi = \langle \mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_n \rangle$, where each instruction $\mathbf{w}_i$ is associated with a program line $0 \leq i \leq n$, and/or a *pointer* z. A pointer $z \in Z$ is a bounded variable with a finite domain $\{0, \ldots, |O|\}$ that indexes an object within the set O. Individual instructions for a planning problem take one of the following forms:

- A planning action $\mathbf{w}_i \in \mathcal{A}$;
- A *goto* instruction $\mathbf{w}_i = \texttt{go}(j, y)$, where j is a program line and y is a fluent value, representing a Boolean condition;
- A *for* instruction $\mathbf{w}_i = \texttt{for}(z_{index}{+}{+}, j)$, which has an associated *endfor* instruction $\mathbf{w}_j = \texttt{endfor}(z_{index}{+}{+}, i)$ (bounding the loop), where $z_{index}{+}{+}$ represents the increment of a specific pointer z_{index}. Such a for loop repeatedly executes the instructions between $i + 1$ and $j - 1$ (inclusive) and iterates over all elements of the collection referred to by the pointer; and
- A termination instruction $\mathbf{w}_i = \texttt{end}$. The last instruction of a planning program Π is always a termination instruction end, i.e., $\mathbf{w}_i = \texttt{end}$ [40].

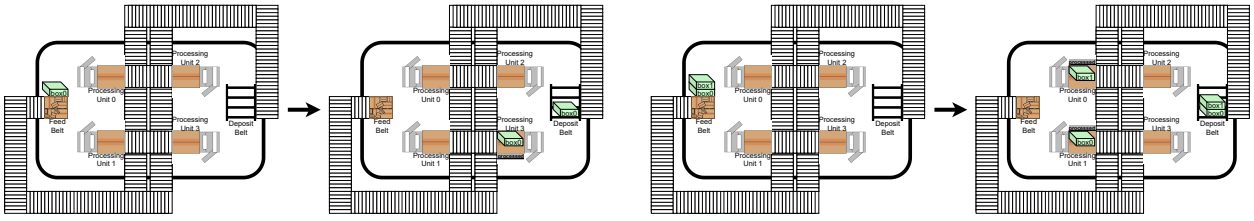
Along with the set of instructions above, planning programs also include *primitive pointer operations* over pointers in Z, such as $\texttt{inc}(z_{index})$, which increments a given pointer by one (up to the maximum index in the collection); $\texttt{dec}(z_{index})$, which decrements a given pointer by one (down to the first index in the collection); $\texttt{clear}(z_{index})$, which sets the given pointer to refer to the first element of the collection; and $\texttt{set}(z_{index}, z_{index'})$, which sets the value of a pointer $z_{index'}$ to the value of another pointer z_{index}.

We define the execution of a generalised plan Π as $exec(\Pi, \mathcal{P}) = \langle a_0, ..., a_n \rangle$, representing an analogy to a plan π, which is a sequence of actions that solves a planning problem $\mathcal{P}$. Since the execution of a generalised plan creates a *linear* classical plan, we will refer to the result of this execution as a *linearisation* of a generalised plan. As defined by Segovia-Aguas et al. [41, 43], the execution context of a generalised plan Π is a program state (s, i, ζ), where s is a planning state $s \in S$, i is the program counter indicating the currently executing line, and ζ is the set of all pointers with their associated current index. Thus, given a planning state and program counter pair (s, i), if $applicable(s, \mathbf{w}_i)$ then the execution of the instruction $\mathbf{w}_i$ results in a new execution state $(s', i + 1)$ where $s' = \mathbf{w}_i(s)$ is the successor state. If $\mathbf{w}_i$ is not applicable, then $s' = s$. If $\mathbf{w}_i = \texttt{go}(j, y)$, the new execution state is (s, j), if y holds in s, and $(s, i + 1)$ otherwise. The proposition value of y can be the result of an arbitrary expression on the state variables, e.g., a state feature. If a for is encountered, then if the associated pointer points to the last element of its collection, the instruction state is updated to point to the next line after the associated endfor. Otherwise, the instruction state points to the following line. When an endfor is encountered, the index of the pointer is incremented. Finally, if $\mathbf{w}_i = \texttt{end}$, execution terminates.

A generalised plan Π solves $\mathcal{P}$ iff the execution terminates in a execution state (s, i) that satisfies the goal condition s_g defined in $\mathcal{P}$, i.e., $\mathbf{w}_i = \texttt{end}$ and $s_g \subseteq s$, otherwise, the execution fails. Thus, for every $\mathcal{P}_i \in \mathcal{GP}, 1 \leq i \leq N$, $exec(\Pi, \mathcal{P}_i)$ solves $\mathcal{P}_i$. We note that the two possible sources of failure of an execution of a generalised plan Π are: (1) an incorrect program, i.e., a generalised plan that terminates in an execution state that does not satisfy the goal condition s_g; and (2) since a generalised plan Π is a program-like plan with control flow commands, an unsound plan could lead to an infinite loop that never reaches an end instruction. This latter failure can be detected by checking duplicate program states. As in Section 2.1, we call $\mathcal{G}\textsc{Planner}(\mathcal{GP})$ an algorithm that solves generalised planning problems.

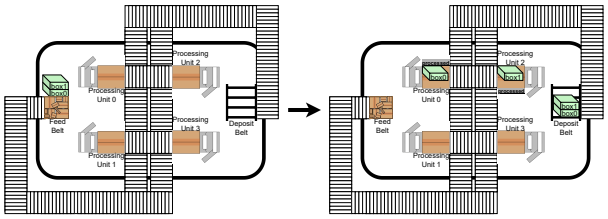
Generalised Planning allows for the injection of domain knowledge through the use of *plan sketches*. These are partially filled general plans, and allow the designer (or some preprocessing step) to specify some operations while leaving others for the planner to discover. Plan sketches can significantly improve planning time, as we discuss in the experiments of Section 5.

To evaluate the resulting architecture, and to illustrate its reasoning process throughout this paper, we adapt the Production Cell scenario from Meneguzzi and Luck [27]. This scenario consists of an abstraction of an automated factory, i.e., the production cell. This cell comprises a number of *processing units* responsible for performing specific manufacturing operations on *blocks* of raw materials depending on the specific part that needs to be produced. Every processing unit is responsible for performing a different kind



(a) Planning problem $\mathcal{P}_0$.

(b) Planning problem $\mathcal{P}_1$.



(c) Planning problem $\mathcal{P}_2$.

Figure 1: Generalised Planning problem example for the Production Cell scenario.

```
0 : for(ptr_block_0++,14)
1 : for(ptr_device_0++,13)
2 : for(ptr_device_1++,12)
3 : for(ptr_procunit_0++,11)
4 : process(ptr_block_0,ptr_procunit_0)
5 : consume(ptr_block_0,ptr_depbelt_0)
6 : move(ptr_block_0,ptr_depbelt_0,ptr_device_0)
7 : move(ptr_block_0,ptr_depbelt_0,ptr_device_1)
8 : move(ptr_block_0,ptr_device_1,ptr_device_0)
9 : move(ptr_block_0,ptr_procunit_0,ptr_device_0)
10 : move(ptr_block_0,ptr_device_0,ptr_device_1)
11 : endfor(ptr_procunit_0++,3)
12 : endfor(ptr_device_1++,2)
13 : endfor(ptr_device_0++,1)
14 : endfor(ptr_block_0++,0)
15 : end
```

Figure 2: Generalised Plan for the Production Cell example.

of operation on a given block, and can process only one block at any given moment. Blocks enter the production cell via a single *feed belt*, and exit the production cell via one or more *deposit belts*, which remove blocks from the production cell to *finish* a block. Conveyor *belts* connect the various parts of the production cell, so the controlling agent needs to move blocks between processing units and the feed and deposit belts.

Figure 1 illustrates the general layout of an instance of the Production Cell scenario with four processing units, two conveyor belts, and one each of a feed belt and a deposit belt. It depicts a generalised planning problem with three different planning problems $\langle \mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2 \rangle$ for the Production Cell domain model, where a robot agent has to move products around locations in the cell while processing them. Figure 2 shows a generalised plan for solving the generalised planning problem depicted in Figure 1, which is composed of three different planning problems.

3 GENERALISED BDI PLANNING

This section formalises the key data structures of GePetto a BDI agent driven by a *Generalised Planning* means-ends reasoner, and specifies the key algorithms underpinning this agent architecture. We follow the presentation style of the surveys on BDI agents from Meneguzzi and de Silva [25] and de Silva et al. [12] while taking inspiration from Pereira and Meneguzzi [33]. Indeed, one can see this paper as providing a pragmatic instantiation of the abstract architecture envisioned by the latter work. The resulting architecture sheds the legacy of reactive planning from AgentSpeak(L)-style languages [35], and focuses on BDI architectures driven by declarative goals [11]. The only assumption we make of the underlying environment is that its dynamics follow a STRIPS-style [16] planning formalism. Finally, we forgo the traditional presentation of BDI architectures using Plotkin's Operational Semantics [34] rules. We do so to provide a more straightforward mapping towards implementation, rather than providing a mathematically elegant presentation.

3.1 BDI Planning

We formalise a BDI interpreter by adapting the formalism of Rao and Georgeff [36]. Here, an agent *Agt* is a tuple $\langle \mathcal{B}, \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{B}$ is a set of *beliefs*, $\mathcal{D}$ represents a set of *desires*, and $\mathcal{I} \subseteq \mathcal{D}$ is a set of adopted *intentions*. Desires are described in terms of environment states the agent can choose to achieve, and intentions as both the desires to which the agent is committed to achieving, and the means by which these will be achieved. Desires are *potential* goals the agent can (but not necessarily does) pursue at any given time. Intentions in turn are those desires that the agent has chosen to actively pursue. In BDI Planning [25], an agent reasons over $\langle \mathcal{B}, \mathcal{D}, \mathcal{I} \rangle$, along with a set of *plan rules* (in a plan-library *Plib*) that represents the behaviours available to the agent.

BDI interpreters can plan via *declarative* or *procedural* planners [25]. BDI planning with declarative planners alleviates the need for a plan-library *Plib* by using declarative goals, much like

Algorithm 1 Planning BDI Reasoning Cycle

Require: Filters DesireFilter, IntentionFilter

Require: Selectors IntentionSelection,

Require: Interfaces Sense, Act, BeliefUpdate, Next

1: **procedure** ReasoningCycle($\mathcal{B}, \mathcal{D}, \mathcal{I}, \Xi$)
2: **loop**
3: $\mathcal{B} \leftarrow$ BeliefUpdate($\mathcal{B}$, sense( ))
4: **if** $\mathcal{I}$ is not empty **then**
5: $\langle\langle\varphi, D\rangle, \Pi_i\rangle \leftarrow$ IntentionSelection($\mathcal{B}, \mathcal{I}$)
6: *result* $\leftarrow$ act(Next($\mathcal{B}, \Pi_i$))
7: **if** Π_i is empty **and** $\mathcal{B} \models D$ **and** *result* $\neq \bot$ **then**
8: ▷ *Intention achieved* ◁
9: $\mathcal{I} \leftarrow \mathcal{I} - \langle\langle\varphi, D\rangle, \Pi_i\rangle$
10: **else if** *result* $= \bot$ **and** $\neg$Retry($\mathcal{B}, \langle\langle\varphi, D\rangle, \Pi_i\rangle$) **then**
11: ▷ *Intention Failed* ◁
12: $\mathcal{I} \leftarrow \mathcal{I} - \langle\langle\varphi, D\rangle, \Pi_i\rangle$
13: **else**
14: $\mathcal{D}_e \leftarrow$ DesireFilter($\mathcal{B}, \mathcal{D}, \mathcal{I}, \Xi$)
15: $\mathcal{I} \leftarrow$ IntentionFilter($\mathcal{B}, \mathcal{D}, \mathcal{I}$)

classical planning, as introduced by the languages GOAL [19], CAN [45], 3APL [10], and 2APL [9].

3.2 Agent Architecture

At its most basic level GePetto comprises a tuple $\langle \mathcal{B}, \mathcal{D}, \mathcal{I} \rangle$. Since we model our environment via STRIPS, we encode beliefs as a database of ground logical fluents $\mathcal{F}$. The agent's sensing function updates $\mathcal{B}$ at each reasoning cycle, which allows the agent to use these beliefs as the initial state for planning, and to keep track of the success (or otherwise) of its actions. Following [33], desires are sets of tuples $\langle \varphi_i, D_i, \sigma_i \rangle$, such that potential goals (D_i) are conjunctive formulas, possibly conditioned by a context formula (φ_i) that restricts when these goals are relevant. Finally, we use an optional preference value σ to allow a designer to indicate an agent's priorities, but omit it (writing $\langle \varphi_i, D_i \rangle$ instead) as appropriate. During the reasoning cycle, the agent chooses desires that satisfy certain criteria, which the agent then stores as part of its intentions $\mathcal{I}$. Intentions focus the agent's reasoning process by narrowing the scope of the often costly process of means-ends reasoning (planning). Thus, an intention is a tuple $\langle\langle \varphi_j, D_j \rangle, \pi_j \rangle$ containing a desire and a plan to achieve such desire. In what follows, we develop the algorithms that drive the BDI reasoning cycle, and define the relationship between these data structures.

3.3 Reasoning Cycle

Algorithm 1 provides a high-level view of the BDI reasoning cycle used by an agent which incorporates a planning algorithm as its means-end reasoner (c.f., [3]). In this section we decompose this high-level algorithm into its component parts to implement the reasoning methodology described in Pereira and Meneguzzi [33], including how we use the generalised planning component within the lower-level algorithms.

Similar to earlier work, an agent updates its beliefs at every cycle (Line 3) and decides whether to continue advancing intentions (if it has any) or reconsider its desires (Lines 4–15). Updating an agent's beliefs comprise a large area of BDI research [1], and we assume a suitable implementation of belief-revision for procedure UpdateBeliefs. This process relies on specific operations on generalised plans, as follows. First, recall from Section 2.2 that the execution (or *linearisation*) of each generalised plan ($exec(\Pi, \mathcal{P})$) yields a classical plan $\langle a_0, ..., a_n \rangle$. Here, in an abuse of notation, we denote the plan Π_i associated to each intention to be either a non-linearised generalised plan, or the steps of a linearisation of the generalised plan for this intention. Thus, function Next($\mathcal{B}, \Pi_i$) carries out the following bookkeeping that comprises an important part of the reasoning cycle. First, it keeps track of previously attempted linearised plans, and, if the generalised plan within an intention does not have a linearised plan, this function tries to generate a new one using as a problem the current state of the belief base and the desire associated with the intention. That is, it tries to find $exec(\Pi_i, \langle \Xi, O, \mathcal{B}, D_i \rangle)$, recalling that a planning problem $\mathcal{P} = \langle \Xi, O, s_0, s_g \rangle$. If there are no new linearisations for the current state of the world, then Next($\mathcal{B}, \Pi_i$) returns a failure, otherwise, the intention now keeps track of the linearised plan, and returns the first action of the linearised plan. If the intention already keeps track of a linearised plan, Next($\mathcal{B}, \Pi_i$) returns the next step of this linearised plan. The agent follows the plan of an intention until either the plan ends its execution successfully, achieving the underlying desire (Line 7), or the agent decides whether to retry the intention (Line 10). Here, Retry($\mathcal{B}, \langle\langle \varphi, D \rangle, \Pi_i \rangle$) implements the decision criteria for retrying an intention while performing bookkeeping with the associated intention. While there are multiple possible implementations of this decision, our current approach consists of trying to generate a new linearisation for the generalised plan associated with the recently failed intention. If $exec(\Pi_i, \langle \Xi, O, \mathcal{B}, D_i \rangle)$ finds no such linearisation for the current belief state, then the agent does not retry the intention. This effectively implements a single-minded commitment, such that the agent only aborts an intention if it can prove that the intention is no longer possible given its beliefs about it. Note that if another generalised plan exists (different from the one associated with the intention during desire filtering), then in the next reasoning cycle, the desire should become eligible again and allow the generalised planner to generate a plan. This effectively makes the agent prioritise intentions for which it has already spent computational resources planning, as linearisation is a cheap operation. We call this approach to managing intentions *late linearisation*, since we defer creating the linear plan to the last moment. The agent then removes successful intentions from its set of intentions (Line 9), leaving the agent free to pursue new intentions. If the agent has achieved all its intentions, it needs to reconsider its active intentions by filtering a set of eligible desires (Line 14), for which the agent can generate a new set of consistent intentions (Line 15).

The reasoning cycle in Algorithm 1 relies on a number of subfunctions that create, filter, and manipulate the various components of the BDI architecture. The first important function, shown in line 1 of Algorithm 2, filters those desires that the agent can commit to achieving from the set of all possible desires. This filtering operates in two steps. First the agent selects desires relevant to the agent's

Algorithm 2 Functions to manage desires and intentions.

1: **function** DesireFilter($\mathcal{B}, \mathcal{D}, \mathcal{I}, \Xi$)
2: ▷ *Relevant Desires* ◁
3: $\mathcal{D}_r \leftarrow \{\langle \varphi, D\rangle \mid \langle \varphi, D\rangle \in \mathcal{D}, \mathcal{B} \models \varphi \wedge \neg D\}$
4: ▷ *Eligible desires* ◁
5: $\mathcal{D}_e \leftarrow \{\langle \varphi, D\rangle \mid \langle \varphi, D\rangle \in \mathcal{D}_r, \exists \pi_D \text{ s.t. } \gamma(\Xi, \mathcal{B}, \pi_D) \models D\}$
6: **return** $\mathcal{D}_e$
7: **function** IntentionFilter($\mathcal{B}, \mathcal{D}_e, \mathcal{I}$)
8: Find $\{\langle \varphi_1, D_1\rangle \dots \langle \varphi_n, D_n\rangle\} \in \mathbb{P}^+(\mathcal{D}_e)$
s.t. $\exists \Pi,\ \Pi = \mathcal{G}\text{Planner}(\{\langle \Xi, \mathcal{B}, D_1\rangle \dots \langle \Xi, \mathcal{B}, D_n\rangle\})$
9: $\mathcal{I} \leftarrow \{\langle\langle \varphi_1, D_1\rangle, \Pi\rangle, \dots \langle\langle \varphi_n, D_n\rangle, \Pi\rangle\}$
10: **return** $\mathcal{I}$
11: **function** IntentionSelection($\mathcal{B}, \mathcal{D}, \mathcal{I}$)
12: ▷ *Filter out intentions whose desires are true* ◁
13: $\mathcal{I} \leftarrow \{I \mid I = \langle\langle \varphi, D\rangle, \Pi_i\rangle \in \mathcal{I}, \mathcal{B} \models \varphi \wedge \neg D\}$
14: ▷ *Check if a linearisation of Π_i plan is executable* ◁
15: $\mathcal{I}' \leftarrow \{I \mid I = \langle\langle \varphi, D\rangle, \Pi_i\rangle \in \mathcal{I}, \mathcal{B} \models pre(\text{Next}(\mathcal{B}, \Pi_i))\}$
16: **return** Pick any intention from $\mathcal{I}'$

current situation (Line 3). These are desires for which the context condition holds in the agent's belief base, and that are not yet satisfied. The agent filters these *relevant desires* further into *eligible desires*, i.e., those that the agent can rationally pursue. We say a desire is eligible if there is a plan of actions that can achieve this desire (Line 5). How an implementation checks for the existence of such a plan can vary. The strongest form of eligibility checking involves running an implementation of a planner to determine whether a desire can be pursued, i.e., find $\pi_D = \text{Planner}(\mathcal{B}, D, \Xi)$. However, practical implementations should use any safe heuristic [32] to rule out unachievable desires. That is, use a heuristic estimate h of the plan cost and rule out desires such that $h(\mathcal{B}, D) = \infty$, which, for admissible heuristics means D is unachievable.

Once the agent has filtered a set of eligible desires $\mathcal{D}_e$, it needs to find all desires which are internally consistent and which can be achieved with a single plan (Line 7). This step is the key difference between GePetto and previous BDI interpreters, including those which utilise automated planning. Specifically, our intention filtering process tries to find a generalised plan for at least one non-empty subset of the power set of eligible desires (Line 8). If one such generalised plan exists, then the agent can use it to drive the achievement of an individual intention attached to each desire (Line 9). Note that an agent's intentions will only be with respect to all those desires for which a *single* generalised plan exists. As detailed in Section 5, there are multiple choices on how to use such generalised plans to drive an agent's behaviour. Finally, once the agent has active intentions, it picks one intention at a time to advance its plan and filters out failed intentions. The intention selection function (Line 11) does so by dropping intentions whose associated desire has already been satisfied, and whose current action is executable (Line 16).

4 THEORETICAL PROPERTIES

In this section, we review the key properties of BDI agents first postulated by Bratman [3], and then logically formalised by Cohen and Levesque [8], as seven key properties of intentions p_i.

(1) Intentions normally pose problems for the agent; the agent needs to determine a way to achieve them.
(2) Intentions provide a "screen of admissibility" for adopting other intentions.
(3) Agents "track" the success of their attempts to achieve their intentions.
(4) The agent believes p_i is possible.
(5) The agent does not believe it will not bring about p_i.
(6) Under certain conditions, the agent believes it will bring about p_i.
(7) Agents need not intend all the expected side-effects of their intentions.

In what follows, we argue that intentions in GePetto naturally enforce these properties in ways that architectures in the style of AgentSpeak(L) do not. GePetto implements Property (1) by construction in IntentionFilter, since each filtered desire becomes the goal condition of a planning problem. Indeed, by extending the definition of planning problem into procedural planning, AgentSpeak(L) architectures adopt tasks as problems in their intention structure, though not declaratively.

The key difference between GePetto and procedural reasoning architectures regarding Property (2) is that there is no natural way to screen parallel intentions for admissibility in the same way as finding a generalised plan for sets of desires/intentions. While work on conflicts for goal-plan trees in BDI agents [44] can detect certain interactions, AgentSpeak(L) style architectures rely on explicit programming of the planning rules to implement similar behaviour.

Tracking the success of an agent's attempts to achieve an intention from Property (3) happens at the level of the main interpreter loop in Algorithm 1. In each reasoning cycle, the agent tracks whether an intention successfully achieves its corresponding desire, dropping an intention that achieves it, or reconsidering the intention in case of failure. In contrast, AgentSpeak(L) style architectures track such progress indirectly, detecting failures if they occur during plan-rule execution. However, such architectures cannot detect — ahead of time — situations where early achievement is potentially achievable, as well as cases where a failure is guaranteed to occur.

Properties (4) to (6) are interrelated, and implemented directly by the filters in DesireFilter and IntentionFilter. Specifically, IntentionFilter generates a proof that a desired property p_i is possible for Property (4) by creating a generalised plan that achieves it under the currently believed conditions c.f., Property (6). Conversely, DesireFilter enforces Property (5) by ruling out desires for which it can find a proof of impossibility (either through a planner, or a safe heuristic). Most AgentSpeak(L)-style architectures lack these key properties without explicit additional programming.

One can argue that the context conditions of AgentSpeak(L)-style approaches enforce Properties (4—6), however, only a few later refinements of this style of architecture actually provide a proof for Property (4) as part of their reasoning process [13]. Finally, planning algorithms naturally satisfy Property (7), as goal conditions in classical planning specify goal formulas, ignoring anything but the explicitly specified goal.

5 EXPERIMENTS AND EVALUATION

5.1 Implementation and Setup

To evaluate GePetto's effectiveness we developed an implementation of this architecture (https://www.meneguzzi.eu/felipe/gepetto). We also developed a variation of the interpreter from Algorithm 1 whose means-ends reasoning process driven by classical planning to use as fair baseline for comparison. We consider this as a fair baseline because the designer specified part of the agent, that is the set of desires with their corresponding preconditions, as well as the environmental description, is exactly the same between the two agents. The key difference between these two agent implementations is that, whereas GePetto IntentionFilter function tries to find a generalised plan for any subset of the desires, the baseline creates a single classical plan for each individual intention, and creates a single intention per desire. Indeed, both interpreters share the same overall infrastructure in terms of agent interface, logic reasoning libraries, and most filters, except for the implementation of the reasoning cycle loop and for the intention filter.

We developed the resulting software packages using Python 3.11. We use BFGP++ from Segovia-Aguas et al. [41] as the generalised planner in GePetto, and rely on a bespoke Python library to generate PDDL files corresponding to the generalised planning problems, as well as the converter from BFGP++ to create generalised planning problems in BFGP++'s formalism. The classical planner in the baseline is a bespoke implementation of a Heuristic Search-based planning [2] algorithm using the Fast Forward heuristic [20]. We ran all experiments in a single core of an Apple Silicon M1 Max CPU with 10 cores and 32GB of RAM.

5.2 Packaging Production Line Scenario

Our first set of experiments is for a series of generalised planning problems for the Packaging Production Line scenario. We develop this scenario to model a warehouse environment that consists of multiple distinct standing units (locations), some of which store packages and boxes. The task involves an agent that needs to collect packages in accordance with their respective quantities and organising them for packaging in boxes. Once packed, they have to be placed at designated locations for dispatch, where they will be prepared for delivery to their corresponding buyers. This scenario captures the complexities of logistical coordination and resource management required in real-world production line and delivery systems, capturing a dynamic environment in which boxes and packages continuously move in and out of the warehouse, creating a fluctuating landscape of resources. Planning becomes particularly challenging as the agent must navigate the task of moving, collecting, and organising these items across multiple standing units (locations). The complexity of this problem is compounded by the need to efficiently coordinate movements and actions to manage inventory at each standing unit, ensuring that packages and boxes are properly organised and prepared for dispatch. The agent's ability to plan effectively in such a dynamic environment is crucial, as it must account for the constant changes and possible uncertainties in the environment. GePetto allows the agent to plan once and use the generalised plan produced (in case one exists) to reason and act in order to navigate through different standing units for collecting boxes and packages to be packed and delivered.

Figure 3 depicts an example of two standing units or locations in a warehouse with packages and boxes that need to be collected and prepared for delivery in specific locations. These are represented through generalised planning problems $\mathcal{P}_0$ and $\mathcal{P}_1$.

We tested GePetto using 15 standing units (locations), varying the number of boxes and packages from 10 to 50. Our simulation generates five sets of generalised planning problems, each defined by different desires (goals) and initial states. The number of problems in each set scales progressively: the first set contains 5 problems with 5 desires, the second 10 problems with 10 desires, and so on, up to 25 problems with 25 desires.

In this scenario, the agent's desires $\mathcal{D}$ take the following form:

$$\langle packed(P_i) \wedge at(P_i, L_i) \rangle$$

where *packed* is predicate that represents that a package P_i is packed, and *at* is a predicate that represents that a package P_i is at location L_i. We ran a simulation of this scenario in a benign setting with no action failures. Thus, the key challenge for the agents is prioritising intentions and generating plans from scratch. In our simulation, we compare the baseline classical planning agent with GePetto, and, given the individual complexity of each problem, set a five-minute time limit for each planner.

We compare two key performance indicators: the number of calls to the planner, and the total runtime efficiency. Both agents generate plans from scratch with no guidance from the designer (i.e., there are no plan sketches to speed up reasoning). Figure 4 summarises these results. First, regarding planner calls, given the lack of errors in the execution, GePetto needs to call the planner a single time once all the packages are in the production line, which Figure 4a highlights. This is so because, in the absence of errors or unsolvable situations, the generalised planner can generate a single generalised plan for all desires at once. Contrast this benign setting with the scenario from Section 5.3. This plan can then be linearised for each intention, resulting in substantial performance gains. In contrast, the classical planning agent needs to generate a linear plan for each relevant desire, at the cost of one planner call per desire. Figure 4b shows this substantial discrepancy in planning time between the two techniques, highlighting the benefits of generalised planning and the GePetto architecture.

5.3 Production Cell Scenario

Our second set of experiments comprises a series of problems continuously generated within a Production Cell scenario containing ten processing units. These problems consist of production runs of an increasingly large number of blocks, each of which requires processing by three random processing units before being finished. Blocks enter sequentially from the feed belt, and a new block enters the production cell whenever the controlling agent removes the previous block from the feed belt. Thus, at any given time, the agent may have multiple blocks within the cell. The production cell is also subject to random outages of processing units, which remain offline for a set amount of time until they come back online. An agent typically detects these errors when it tries to move a block into a processing unit, which requires it to replan, thus creating a more challenging setting for the individual agents. Indeed, this scenario is so complex (for a generalised planning problem) that the runtime cost of generating a generalised plan from scratch is

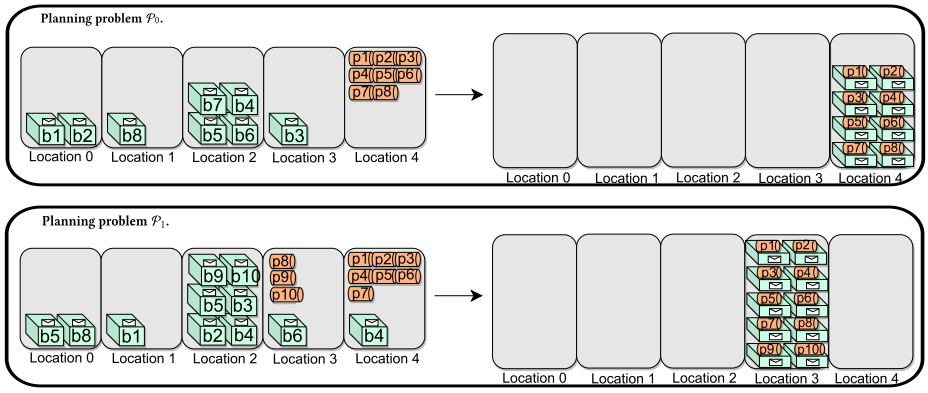
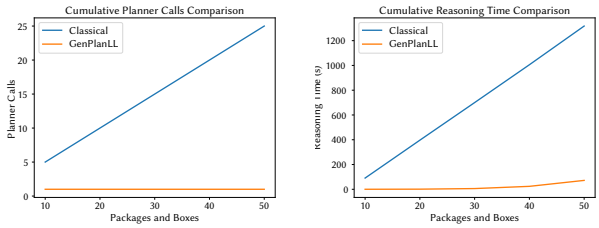


Figure 3: Generalised Planning problem example for the Packaging Production Line scenario.



(a) Contrasting Planner calls. **(b) Total Reasoning time.**

Figure 4: Empirical results for Packaging Production Line.

orders of magnitude worse than generating classical plans piecemeal. This illustrates how plan sketches provide a pragmatic BDI architecture even when generalised planning from scratch is not efficient, allowing domain knowledge to be introduced as needed.

In this scenario, the agent's desires $\mathcal{D}$ are of the form:

$$\langle over(B, F) \wedge \bigwedge_{i=1}^{k} requires(B, Pu_i),$$
$$finished(B) \wedge \bigwedge_{i=1}^{k} processed(B, Pu_i)\rangle$$

Thus, when a block (B) enters the production cell through feedbelt F with a set of requirements for processing, the agent should create a desire to have this block processed by the required processing units (Pu_i) and finished. We conducted experiments with a range of 10 to 50 blocks introduced into the production cell, an error rate of 0.2, and errors lasting for six time steps. This means that, at every turn, there is a 20% chance of a random processing unit having a fault, and this unit only becomes available again after 6 time steps. Only one processing unit can be faulty at a time. Thus, if any block currently in the production cell depends on a faulty processing unit, this presents the agent with two possible challenges. Either the plan to satisfy this desire will fail (if the agent had already created a plan to finish the block), or the means-ends reasoner should conclude that this desire is not currently achievable.

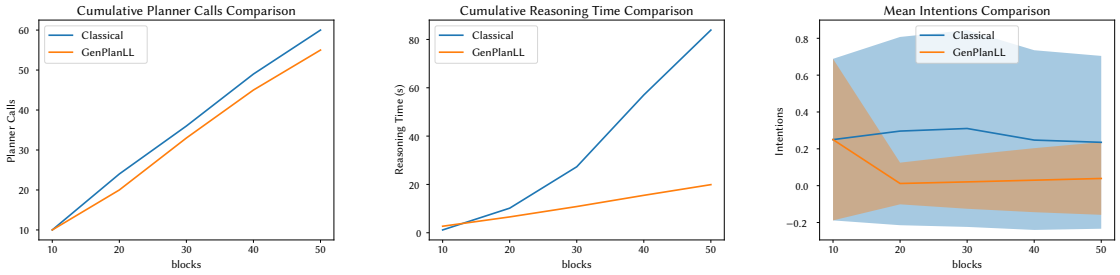
Figure 5 illustrates the qualitative results for this experiment. Figure 5a contrasts the cumulative number of times each agent architecture calls the planner throughout the simulation. One side effect of potential errors throughout the simulation is that, as errors occur that prevent blocks from being processed, the number of relevant desires increases as more blocks remain unfinished in the production cell. This shows that as unfinished blocks occasionally accumulate and the means ends reasoner needs to find plans to finish multiple desires in a single reasoning cycle, using a generalised planner allows the agent to make fewer calls to the planner, since each call can deal with multiple desires at the same time. For example, if there are three unfinished blocks in the production cell, and there exists a single generalised plan that can finish all three blocks, this will result in a single planner call, whereas the classical planner would need at least three separate calls. The overall effect, for this scenario, is that the generalised planning-driven agent always requires fewer planner calls.

The second notable improvement of a generalised planning-driven agent that uses plan sketches, is that the overall reasoning time is substantially lower. Figure 5b illustrates the runtime efficiency gained by using a cached generalised plan sketch. In this case, we ran BFGP++ offline to generate a generalised plan sketch, and always try to plan using BFGP++'s option to repair a plan sketch. This yields dramatic improvements in runtime performance over the classical planner, effectively bridging the performance gap between the reasoning based on plan-rules from earlier architectures, and the flexibility of classical planning.

Finally, we look at the evolution of the number of active intentions over time in Figure 5c. This shows that, as the classical planner needs to deal with failures one at a time, when unfinished blocks accumulate, the number of intentions the agent needs to clear remains consistently high throughout the simulation.

6 RELATED WORK

We now contrast our contributions with recent work on automated Planning for BDI-style agents. First, we discuss BDI-based architectures that rely on a planning algorithm to generate sequential plans without plan rules. Second, we discuss BDI-style architectures that use a planner as a look-ahead mechanism for intention selection



(a) Contrasting Planner calls. **(b) Total Reasoning time.** **(c) Intentions backlog over time.**

Figure 5: Empirical results for the Production Cell scenario.

even when the agent ultimately generates plans using reactive planning rules. Finally, we describe other recent agent architectures that carry out planning and acting backed by a planning algorithm.

Regarding classical Planning in BDI agents, while there is a long tradition of incorporating various forms of such planning into the reasoning cycle of BDI-style agents [6, 25], we compare to a few classical architectures. Most related to this architecture is the work of Meneguzzi et al. [29], which describes a logic-based BDI agent architecture that creates intentions using a classical planner. A related effort [26] introduces an explicit planning action into AgentSpeak(L) that allows a designer to add plan-rules that convert reactive planning events into calls to an external classical planner. More recently, Zanetti et al. [47] extends the ROS2-BDI framework with a classical planner to continually generate plans of action for the agent in a robotic simulation. This architecture is roughly equivalent to the baseline we use in the experiments. Finally, Xu et al. [46] develop a more advanced, and formally grounded, extension to AgentSpeak(L) planning operator from Meneguzzi and Luck [26].

Besides generating new plans from scratch, various AgentSpeak(L)-like BDI architectures have used the analogous nature of HTN and BDI planning to perform lookahead on potential rollouts of rules. Researchers [13, 39] formalised extensions to the JACK [21] programming language that uses an HTN planner [23] to simulate the progress of its planning rules. Similarly, [38] formalise the behaviour of an AgentSpeak(L) style agent reasoning cycle using a plan function to simulate the possible outcomes of planning rules.

Outside the BDI tradition, the planning community developed various planning and action approaches. While not exactly a single agent architecture like BDI, MAPL [5] provides both a programming language and an execution framework for distributed multiagent planning. Most recently, Patra et al. [31] developed a planning and acting [30] architecture that is closely related to BDI-style agent architectures that try to perform lookahead when deciding its courses of action. However, instead of using an HTN planner to find the most promising decomposition like the efforts in BDI architectures [13, 38, 39], their architecture uses Monte Carlo simulations to account for potential errors in the environment.

7 DISCUSSION AND CONCLUSIONS

This paper instantiates the conceptual architecture of our Blue Sky paper [33] by introducing GePetto, the first concrete BDI architecture whose means-ends reasoning process relies entirely on generalised Planning. Besides the ability to automatically generate plans at runtime as previous planning [6, 25, 46, 47] BDI architectures have done in the past, the resulting architecture provides key advantages both theoretical and practical. On the theoretical side, GePetto automatically enforces the seven key properties of intentions as postulated by the foundational paper from Cohen and Levesque [8]. On the practical side, we empirically show in two separate scenarios that a BDI architecture driven by generalised planning, under certain conditions, is more efficient in terms of planner calls and of reasoning time. Indeed, this is thanks to recent advances in generalised Planning algorithms [42] that underpin many of the empirical advantages GePetto shows. These are scaled-up heuristic search, and the possibility to bootstrap search with plan sketches. This latter capability allows a designer to use a generalised planner offline, or manually design, plan sketches that may substantially reduce the runtime cost of running the planner. Given the fast pace at which research on generalised planning has progressed, we expect that even in problems that necessitate offline plan-sketch generation, we may see improvements soon. Thus, GePetto combines strong theoretical guarantees with a convenient trade-off between the computational efficiency of traditional BDI architectures and the flexibility of purely planning-driven ones.

GePetto is the first of what is an entirely new family of BDI generalised planning architectures. As such, it implements a relatively simple agent reasoning cycle. For example, its strategy for failure recovery consists of re-linearising a generalised plan, or eventually retrying planning from scratch. It also does not incorporate reasoning about social aspects of a multiagent system, including norms [14], communication and commitment protocols [28]. This remains as future work. Besides the algorithmic aspects of this family of agent architecture, future work will investigate designing and deploying GePetto in practical multiagent systems.

This paper bridges the gap between two historically separate but pragmatically interrelated areas, namely autonomous agents and automated planning. GePetto, thus provides a platform for both areas to experiment, expand, and integrate associated insights.

ACKNOWLEDGMENTS

We thank Javier Segovia-Aguas for help on Generalised Planning and BFGP++. This study was part funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

REFERENCES

[1] Natasha Alechina, Mark Jago, and Brian Logan. 2005. Resource-Bounded Belief Revision and Contraction. In *Declarative Agent Languages and Technologies III, 3rd International Workshop (Lecture Notes in Computer Science, Vol. 3904)*, Matteo Baldoni, Ulle Endriss, Andrea Omicini, and Paolo Torroni (Eds.). Springer, 141–154.

[2] Blai Bonet and Héctor Geffner. 2001. Planning as heuristic search. *Artificial Intelligence* 129, 1 (2001), 5–33.

[3] Michael E Bratman. 1984. Two Faces of Intention. *Philosophical Review* 93 (1984), 375–405.

[4] Michael E Bratman, David J Israel, and Martha E Pollack. 1988. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence* 4, 4 (1988), 349–355.

[5] Michael Brenner and Bernhard Nebel. 2009. Continual planning and acting in dynamic multiagent environments. *Auton. Agents Multi Agent Syst.* 19, 3 (2009), 297–331.

[6] Rafael C. Cardoso, Angelo Ferrando, and Fabio Papacchini. 2021. Automated Planning and BDI Agents: A Case Study. In *Advances in Practical Applications of Agents, Multi-Agent Systems (PAAMS)*, Vol. 12946. 52–63.

[7] Sergio Jiménez Celorrio, Javier Segovia-Aguas, and Anders Jonsson. 2019. A Review of Generalized Planning. *Knowledge Engineering Review* 34 (2019), e5.

[8] Phillip R Cohen and Hector J Levesque. 1990. Intention is choice with commitment. *Artificial Intelligence* 42, 2-3 (1990), 213–261.

[9] Mehdi Dastani. 2008. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems* 16, 3 (2008), 214–248.

[10] Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer. 2003. A Programming Language for Cognitive Agents Goal Directed 3APL. In *Programming Multi-Agent Systems, First International Workshop (PROMAS)*. 111–130.

[11] Mehdi Dastani, M Birna van Riemsdijk, and Michael Winikoff. 2011. Rich goal types in agent programming. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 405–412.

[12] Lavindra de Silva, Felipe Meneguzzi, and Brian Logan. 2020. BDI Agent Architectures: A Survey. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. International Joint Conferences on Artificial Intelligence Organization, 4914–4921.

[13] Lavindra de Silva, Sebastian Sardina, and Lin Padgham. 2009. First principles planning in BDI systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*. 1105–1112.

[14] Davide Dell'Anna, Natasha Alechina, Fabiano Dalpiaz, Mehdi Dastani, Maarten Löffler, and Brian Logan. 2022. The Complexity of Norm Synthesis and Revision. In *Coordination, Organizations, Institutions, Norms, and Ethics for Governance of Multi-Agent Systems (Lecture Notes in Computer Science, Vol. 13549)*, Nirav Ajmeri, Andreasa Morris-Martin, and Bastin Tony Roy Savarimuthu (Eds.). Springer, 38–53.

[15] Kutluhan Erol, James Hendler, and Dana S Nau. 1994. HTN Planning: Complexity and Expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. 1123–1128.

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

[17] Hector Geffner and Blai Bonet. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publishers.

[18] Malte Helmert. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26 (2006), 191–246. https://www.fast-downward.org

[19] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. 2001. Agent Programming with Declarative Goals. In *Proceedings of the 7th International Workshop on Intelligent Agents, Agent Theories Architectures and Languages*. Springer-Verlag, 228–243.

[20] Jörg Hoffmann and Bernhard Nebel. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14 (2001), 253–302.

[21] Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. 2001. JACK Intelligent Agents: Summary of an Agent Infrastructure. In *Proceedings of the 5th International Conference on Autonomous Agents*.

[22] Yuxiao Hu and Giuseppe De Giacomo. 2011. Generalized Planning: Synthesizing Plans that Work for Multiple Environments. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 918–923.

[23] Okhtay Ilghami and Dana S Nau. 2003. *A General Approach to Synthesize Problem-Specific Planners*. Technical Report.

[24] François Félix Ingrand and Vianney Coutance. 2001. *Real-Time Reasoning Using Procedural Reasoning*. Technical Report 93104. LAAS/CNRS. Technical Report.

[25] Felipe Meneguzzi and Lavindra de Silva. 2015. Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. *Knowledge Engineering Review* 30, 1 (2015), 1–44.

[26] Felipe Meneguzzi and Michael Luck. 2008. Leveraging new plans in AgentSpeak(PL). In *Proceedings of the Sixth Workshop on Declarative Agent Languages*. 63–78.

[27] Felipe Meneguzzi and Michael Luck. 2013. Declarative planning in procedural agent architectures. *Expert Systems with Applications* 40, 16 (2013), 6508 – 6520.

[28] Felipe Meneguzzi, Pankaj R. Telang, and Munindar P. Singh. 2013. A First-Order Formalization of Commitments and Goals for Planning. In *AAAI Conference on Artificial Intelligence*, Marie desJardins and Michael L. Littman (Eds.). 697–703.

[29] Felipe Meneguzzi, Avelino F Zorzo, and Michael C Mora. 2004. Mapping Mental States into Propositional Planning. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*. 1514–1515.

[30] Sunandita Patra, James Mason, Malik Ghallab, Dana Nau, and Paolo Traverso. 2021. Deliberative acting, planning and learning with hierarchical operational models. *Artificial Intelligence* 299 (2021), 103523.

[31] Sunandita Patra, James Mason, Amit Kumar, Malik Ghallab, Paolo Traverso, and Dana Nau. 2021. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. *Artificial Intelligence* 30 (2021), 478–487.

[32] Judea Pearl. 1984. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley.

[33] Ramon F. Pereira and Felipe Meneguzzi. 2024. Empowering BDI Agents with Generalised Decision-Making. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. 2679–2683.

[34] Gordon D Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report. University of Arhus.

[35] Anand S Rao. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world*, Vol. 1038. Springer-Verlag, 42–55.

[36] Anand S. Rao and Michael P. Georgeff. 1995. BDI Agents: From Theory to Practice. In *Proceedings of the First International Conference on Multiagent Systems*. 312–319.

[37] Scott E. Reed, Konrad Zolna, Emilio Parisotto, Sergio Gómez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. 2022. A Generalist Agent. *Transactions on Machine Learning Research* 2022 (2022).

[38] Sebastian Sardina and Lin Padgham. 2011. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems* 23, 1 (2011), 18–70.

[39] Sebastian Sardiña, Lavindra de Silva, and Lin Padgham. 2006. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*. 1001–1008.

[40] Javier Segovia-Aguas, Sergio Jiménez Celorrio, and Anders Jonsson. 2019. Computing Programs for Generalized Planning using a Classical Planner. *Artificial Intelligence* 272 (2019), 52–85.

[41] Javier Segovia-Aguas, Sergio Jiménez Celorrio, and Anders Jonsson. 2022. Computing Programs for Generalized Planning as Heuristic Search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 5334–5338.

[42] Javier Segovia-Aguas, Sergio Jiménez Celorrio, Laura Sebastiá, and Anders Jonsson. 2022. Scaling-Up Generalized Planning as Heuristic Search with Landmarks. In *Proceedings of the International Symposium on Combinatorial Search (SOCS)*. 171–179.

[43] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. 2021. Generalized Planning as Heuristic Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 569–577.

[44] John Thangarajah, Lin Padgham, and Michael Winikoff. 2003. Detecting & Avoiding Interference Between Goals in Intelligent Agents.. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 721–726.

[45] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. 2002. Declarative & Procedural Goals in Intelligent Agent Systems. In *Proceedings of the Eight International Conference on Principles and Knowledge Representation and Reasoning (KR)*. 470–481.

[46] Mengwei Xu, Tom Lumley, Ramon Fraga Pereira, and Felipe Meneguzzi. 2024. [PDF] from meneguzzi.eu A Practical Operational Semantics for Classical Planning in BDI Agents. In *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI)*.

[47] Alex Zanetti, Devis Dal Moro, Redi Vreto, Marco Robol, Marco Roveri, and Paolo Giorgini. 2023. Implementing BDI Continual Temporal Planning for Robotic Agents. In *IEEE International Conference on Web Intelligence and Intelligent Agent Technology*. 378–382.