Improving domain-independent intention selection in BDI systems

Article *in* Autonomous Agents and Multi-Agent Systems · July 2015

DOI: 10.1007/s10458-015-9293-5

Max Waters
RMIT University
2 PUBLICATIONS **12** CITATIONS
SEE PROFILE

Lin Padgham
RMIT University
210 PUBLICATIONS **3,918** CITATIONS
SEE PROFILE

Sebastian Sardina
RMIT University
91 PUBLICATIONS **989** CITATIONS
SEE PROFILE

Noname manuscript No.
(will be inserted by the editor)

Improving Domain-Independent Intention Selection in BDI Systems

Max Waters · Lin Padgham · Sebastian Sardina

Received: date / Accepted: date

Abstract The Belief Desire Intention (BDI) agent paradigm provides a powerful basis for developing complex systems based on autonomous intelligent agents. These agents have, at any point in time, a set of intentions encoding the various tasks the agent is working on. Despite its importance, the problem of selecting *which intention to progress* at any point in time has received almost no attention and has been mostly left to the programmer to resolve in an application-dependent manner. In this paper, we implement and evaluate two *domain-independent* intention selection mechanisms based on the ideas of *enablement checking* and *low coverage prioritisation*. Through a battery of automatically generated synthetic tests and one real program, we compare these with the commonly used intention selection mechanisms of *First-In-First-Out* (FIFO) and *Round Robin* (RR). We found that enablement checking, which is incorporated into low coverage prioritisation, is never detrimental and provides substantial benefits when running vulnerable programs in dynamic environments. This is a significant finding as such a check can be readily applied to FIFO and RR, giving an extremely simple and effective mechanism to be added to existing BDI frameworks. In turn, low coverage prioritisation provides a significant further benefit.

Keywords BDI Agent Programming · Intention Selection

Max Waters
School of Computer Science & Information Technology
RMIT University
Melbourne, AUSTRALIA
E-mail: max.waters@rmit.edu.au

Lin Padgham
School of Computer Science & Information Technology
RMIT University
Melbourne, AUSTRALIA
E-mail: lin.padgham@rmit.edu.au

Sebastian Sardina
School of Computer Science & Information Technology
RMIT University
Melbourne, AUSTRALIA
E-mail: sebastian.sardina@rmit.edu.au

1 Introduction

This work is concerned with the important problem of *intention selection* in intelligent agent systems. Intelligent autonomous agents are expected to be able to act appropriately in complex dynamic environments to achieve their overall goals. The Belief Desire Intention (BDI) paradigm [6, 9, 23, 24] and associated programming languages and implemented toolkits [2, 28] are amongst the most successful approaches to realising smart autonomous agents. Under such a paradigm, the agent's behavior arises from its intentions, the current set of objectives that the agent has committed to realize—the agent's focus of attention. In computationally grounded BDI systems, such intentions resemble plans that are meant to achieve a certain goal. As agents will, in general, pursue multiple intentions, a great deal of the power of the paradigm relies on the mechanisms for selecting *which* intention to focus on at any particular point in time. Indeed, the core reasoning tasks involved in rational architectures —including deliberation, filtering, and means-end analysis [6, 24]—are generally integrated, to some extent, into the way the agent chooses the intention to execute next.

Unfortunately, the mechanisms for intention selection generally supported by typical BDI infrastructures are very simplistic. Most existing BDI platforms only provide rudimentary intention scheduling mechanisms such as First-In-First-Out (FIFO), Round Robin (RR), or simple priority-based schedulers. As a result, anything more "intelligent" than that must be explicitly programmed in an application-dependent way, such as using semaphores to synchronise a set of intentions, atomic blocks to force transaction-like executions, or even complete meta-programming of the intention selection task in a specific language as in 3APL [16]. There are other BDI-like systems that provide more sophisticated built-in intention selection mechanisms by making use of additional information given by the domain expert, such as deadlines, priorities, inter-relationships amongst plans, plan cost and quality, etc. (e.g., [4, 19, 34]). However, all such approaches require specific domain-dependent information that may not be available, and so the question remains: *how much support for intention selection can a BDI infrastructure provide without imposing extra requirements on the programmer?*

In this work, we describe and empirically evaluate two *domain-independent* approaches to more intelligent intention selection, initial versions of which were described in conference papers [33] and [35]. The approaches described rely entirely on information already present in standard BDI programs, and thus require neither extra information nor additional programming from the developer. We call the two approaches *enablement checking* and *low coverage prioritisation*. Enablement checking is based on the intuition that in general, an agent should not attempt to progress an intention if it knows that there is currently no way to accomplish the next step. Low coverage prioritisation captures the idea that in addition to this, if there are intentions for which there is "weak" know-how available, then an agent should prioritise pursuing such intentions when the situation is amenable to such pursuit. That is, "vulnerable" intentions should take priority when they are currently able to be progressed. To capture that, we rely on the agent-oriented software engineering [22] notion of plan *coverage* (and overlap) to capture the relative robustness/vulnerability of a given intention.

We have implemented three new intention selection mechanisms within Jack [7], two of which add enablement checking to the existing mechanisms of FIFO and RR, and one which implements low coverage prioritisation. We empirically evaluate these new selection mechanisms by comparing them to FIFO and RR over a large number of synthetic, automatically generated programs running in a simulated environment with varying levels of dynamism. We also empirically evaluate enablement checking on a real program (an extended Tower of Hanoi), examining concepts such as fairness and efficiency, in addition to the number of successfully completed intentions.

In our experiments, we found that low coverage prioritisation results in the proportion of successfully completed intentions increasing by up to 64.9 (FIFO) and 67.8 (RR) percentage points, in volatile environments, where the plan library contains significant know-how gaps. Perhaps more importantly we found that the extremely straightforward enablement check provides the largest part of this benefit. This is significant because such a check can be incorporated into FIFO and RR within existing BDI systems in a straighforward manner. In Section 5, we confirm these results using a more complex, real program. We found that enablement checking improves the efficiency, fairness, and success rate of FIFO, and that RR receives an important increase in efficiency and success rate with minimal degradation in fairness. Overall, the results support the integration of enablement checking into standard scheduling mechanisms and open the door for even more sophisticated domain-independent techniques based on the quality of the know-how available to the agent.

The remainder of the paper is organised as follows. We first provide some background on the structure of BDI programs and the problem of intention selection, including a definition of the representation of intentions which we will use. We then describe in Section 3 our enablement checking and low coverage prioritisation approaches, and the selection mechanisms tested. Section 4 describes our evaluation on synthetic domains, and Section 5 describes the evaluation on the extended multiple Towers of Hanoi program. We conclude with related work followed by a discussion including directions for future work.

2 The Intention Selection Task

BDI agent-oriented programming is a popular, well-studied, and practical-oriented paradigm for building intelligent agents situated in complex and dynamic environments with (soft) real-time reasoning and control requirements [1, 13]. Besides its philosophical roots in practical reasoning [6, 23] and theoretical understandings [9, 24], there are many BDI-style programming languages and systems available, such as Jack, Jason, Jadex, 2APL, and GOAL [2, 11, 28].

A typical BDI-style agent system is depicted in Figure 1. An agent consists of a *belief base* encoding the agent's knowledge about the world (akin to a database in most real systems), a set of recorded *events* or *goals* that are pending to be addressed by the agent, a *plan library*, and an *intention base*.[1] We explain the latter two components in some detail, since they are central to the topic of this article.

[1] As customary in agent programing, we use the terms *event* and *goal* interchangeably. This is due to seeing goals procedurally as "goals-to-do" (i.e., respond to events), rather that the alternative "goals-to-be"

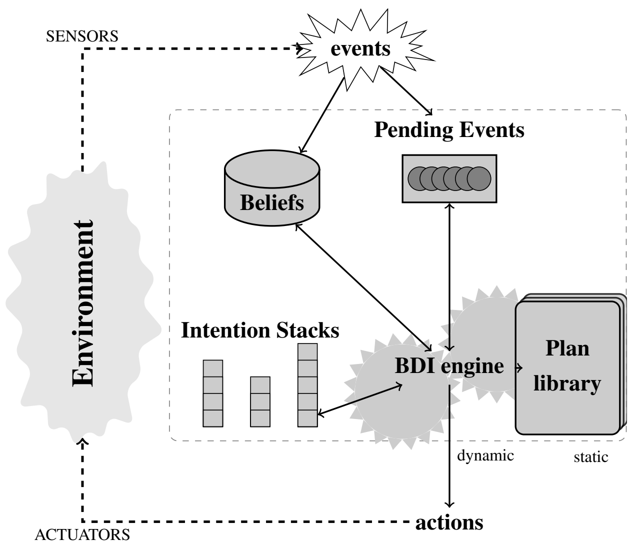


Fig. 1 A typical BDI Agent Programming Framework.

BDI plans & plan library The *plan library* contains the typical operational procedures of the domain indicating how the system may respond to event-goals. Concretely, a BDI plan library amounts to a set of rules of the form $e : \psi \leftarrow \delta$ with the intended meaning that *"program δ is a reasonable strategy for achieving/resolving event-goal e when context condition ψ is believed true."* A plan body procedure δ will typically be built from (the execution of) domain actions in the environment (e.g., opening the elevator's door) and the posting of additional subgoals $!G$ (e.g., move the elevator to a given floor) that are in turn resolved by selecting suitable plans. For simplicity, in this paper we consider only sequences in plan bodies. For example, the following plan rules may be part of the plan library of an elevator controller:

$$\begin{aligned} \mathit{RequestOn}(\mathit{floor}) : \mathit{Serving}(\mathit{floor}) &\leftarrow !\mathit{GoTo}(\mathit{floor});\ \mathit{Open}; \mathit{Close}; \mathit{Off}(\mathit{floor}) \\ \mathit{GoTo}(\mathit{floor}) : \mathit{At}(x) \wedge x > \mathit{floor} &\leftarrow \mathit{GoUp};\ !\mathit{GoTo}(\mathit{floor}) \end{aligned}$$

The elevator controller can address a floor request (i.e., goal $\mathit{RequestOn}(\mathit{floor})$) whenever the elevator is in charge of serving that floor (i.e., when $\mathit{Serving}(\mathit{floor})$ is true) by first achieving the subgoal of going to the floor in question (i.e., resolving subgoal $!\mathit{GoTo}(\mathit{floor})$), then opening and closing the door, and finally turning the floor's request light off. The second plan rule states how to navigate upwards.

Goal-plan trees As explained above, the behavior of a typical BDI agent system arises from the system responding to event-goals by resorting to the know-how information in its plan library. In doing so, a plan rule may be selected for addressing an event if it is *relevant* and *applicable*, that is, if it is a plan designed for the event in

perspective taken in agent theory [9, 24], for example. Nonetheless, we recognise the existence of other approaches to BDI programming with a more declarative perspective on goals.

question and its context condition is believed true. In contrast with traditional planning, execution happens at each step. The assumption is that the use of plans' context preconditions to make choices as late as possible, together with the built-in goal-failure mechanisms, ensures that a successful execution will eventually be obtained while the system remains sufficiently responsive to changes in the environment.

An important notion for this paper is that of a goal-plan tree induced by the agent's plan library. By grouping together all of the plans which respond to the same event type, the plan library induces *goal-plan tree* templates of the sort shown in Figure 2(a) (ignore numeric annotations). An event-goal node (e.g., goal G_1) has child nodes representing the alternative plans for achieving it under different circumstances (i.e., plans P_1 and P_2 can achieve G_1 when proposition p is believed true or false, respectively). Plan nodes, in turn, have child nodes representing the subgoals (including primitive actions) of the plan (i.e., subgoals G_2 and G_1 in plan P_1). These structures can be seen as AND/OR trees: for a plan to succeed, all of the subgoals and actions of the plan must be successful (AND); for a subgoal to succeed at least one of the plans to achieve it must succeed (OR). Leaf plans (e.g., P_3 to P_8) include no subgoals, but only primitive actions.

Intentions The *intention base* $\mathcal{I}$ stands for the set of current (partially executed) plan-programs that the agent has already committed to execute in order to resolve some of the pending goals. As is customary in the semantics of programming languages with procedure invocation, an intention I is generally represented as a *stack* of the form $I = [(G_n : \delta_n)(G_{n-1} : \delta_{n-1}) \cdots (G_0 : \delta_0)]$, with $n \geq 0$, where:

- G_0 is the top-level (external) event-goal—the original reason for intention I;
- δ_i is what remains to be executed from the plan selected to address (sub)goal G_i;
- G_i, for each $n \leq i \leq 1$, is an "active" subgoal in program δ_{i-1}; and
- G_n is the most current active subgoal of the intention, that is, the goal the agent is currently working on by carrying out plan-program δ_n.

Almost all BDI systems realize, in one way or another, the basic abstract rational architectures described by Rao and Georgeff [26] and Bratman et al. [6]. There, the rational reasoner regularly performs three key decision-making tasks (c.f. Figure 1):

1. *select one or more pending event-goals* to handle (deliberation and filtering);
2. *select plans* to handle such goals and commit to them (means-end reasoning); and
3. *select one or more intentions* and (partially) execute them (execution).

Together, these three decision-making tasks provide the actual "intelligence" of the whole approach. In this work, we are concerned with the third decision, *how to select which intentions to advance next at a given point*. However, when it comes to computationally realizing these three selection tasks, most agent architectures often combine them, at least to some extent. For example, deciding which plan to use to resolve an (internal) pending event-goal is often seen as a special execution step in an intention. Similarly, deliberating over which event to handle next and which intention to execute, are almost always seen as a joint decision (except for external events).

So, to be more precise, a typical BDI architecture realizes the above three decision-making steps by performing the following tasks (often jointly in various ways):

1. *select a pending event-goal* G to handle, where G is either *external* (i.e., has been posted as a result of a percept update, message received, etc.), or *internal* (i.e., has been posted as a subgoal of one of the agent's intentions);
2. *select a plan rule* $G : \phi \leftarrow \delta$ from the plan library and commit to plan δ:
   - If G is external, a new intention $I = [(G : \delta)]$ is added to the intention base.
   - If G is internal and arising from intention $I = [(G_n : !G; \delta_n) \cdots (G_0 : \delta_0)]$, then the intention is updated to $I = [(G : \delta)(G_n : \delta_n) \cdots (G_0 : \delta_0)]$.
3. *select an intention* I (from the agent's current intention base $\mathcal{I}$) to be advanced, the execution of which could result in the performance of some domain action or the generation of a new internal event-goal via a subgoal posting step $!G$.

(For examples on detailed formal semantics realizing the typical BDI rational execution cycle, we refer the reader to [3, 24, 28].)

At every step in the execution of an agent system, the intention base $\mathcal{I}$ represents the different focuses of attention the system has in order to respond to those events that the agent has committed to address [6]. *The overarching objective is for the system to be successful in carrying out all those intention programs to completion, thus resolving the corresponding events.* A problem, though, is that some such programs may end up failing because their execution context changes in unexpected ways. Indeed, a program δ may post a subgoal $!G$ for which the agent has no adequate plan in its library. This could happen due to changes in the environment or due to undesirable interactions among the various executing intentions. If the agent is able to intelligently address its intentions when the situation for doing so matches its know-how, the chances of success will be improved.

Despite this problem of *intention selection* being at the core of the BDI approach to agency, it has, unfortunately, been little studied, with solutions being either extremely simplistic or reliant on domain-specific programming and non-trivial extra information to achieve the desired control.

Almost all existing agent platforms, such as Jack [7] or Jason [5], offer a choice between RR, which does a fixed number of steps on each intention in turn, or a FIFO queue, which processes each intention in the order received, moving it to the back of the queue if, for some reason, it suspends. Many languages and platforms also provide facilities to program (application-specific) intention selection strategies [7, 16]. There have also been proposals to enhance intention selection by taking into account goal deadlines (e.g., AgentSpeak(RT) [34]), and priorities or utilities on goals and/or plans (e.g., JAM [19] and AgentSpeak(XL) [4]). However, all of these approaches require the developer to either explicitly program the intention scheduler, or provide additional domain information, which is often not readily available. Here, on the other hand, we explore domain-independent techniques that require neither.[2]

3 Domain-independent Intention Selection

The two standard domain-independent intention selection approaches found in BDI platforms are FIFO and RR. FIFO represents the intention base as a queue, and always

[2] Such techniques can still be integrated with domain-specific schemes, see the discussion in Section 7.

selects the intention at the front of it, keeping focus on this intention until it either ends (passes or fails) or is suspended for some reason, in which case it is moved to the back of the queue. RR, on the other hand, cycles across all intentions, spending an equal amount of effort (time-slice, quantum, or just number of steps) on each before moving on to the next.

Both FIFO and RR are appealing for many reasons: they are easy to implement, require little in the way of computational cost and require no extra information from the programmer. However, they are not particularly "intelligent." RR is a "fair" scheme, with low response times and hence truly realises the principle that intelligent agents have several focuses of attention at any point in time [6]. However, its interleaving of intentions increases the possibility of failure due to undesirable interference. On the other hand, FIFO maintains focus on a single task, thus reducing the possibility of failure due to interference. As a result, it tends to increase the number of successfully executed intentions at the expense of response times and fairness.

Neither FIFO nor RR incorporate any level of reasoning regarding which intention is best to pursue at a given time, or how to limit interference between intentions. Such reasoning is left to the programmer to address in a domain-specific way, generally by programming synchronisation points within BDI plans or developing application-specific intention selection schemes.

Our aim is to develop intention selection mechanisms which are more powerful than FIFO and RR, *without requiring additional design knowledge or programming effort* beyond that which is found in standard BDI programs. Though not the topic of this paper, we also aim to obtain techniques that are simple and flexible enough to be coupled with domain-dependent schemes (see Section 7). To that end, we present below two generic domain-independent intention selection techniques, namely, *enablement checking* and *low coverage prioritisation*. In the remainder of the paper, we shall evaluate these by comparing them with FIFO and RR.

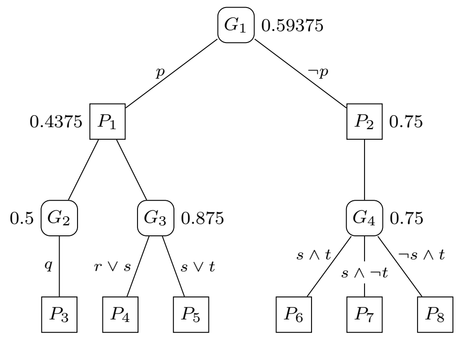
3.1 Enablement checking

Enablement checking is the simple yet powerful idea of determining whether or not an intention's next step will be to post a subgoal with no applicable plans in the current state. By examining the context conditions of plans relevant to a subgoal, it can readily be determined whether there is some applicable plan (i.e., a plan whose context condition evaluates to *true*) in the current context. If the next step in an intention is either a primitive action or a subgoal with an applicable plan, then the intention is regarded as *enabled*. If not, the intention is not enabled and may not be selected for progression at the current time (unless no intention is enabled[3]).

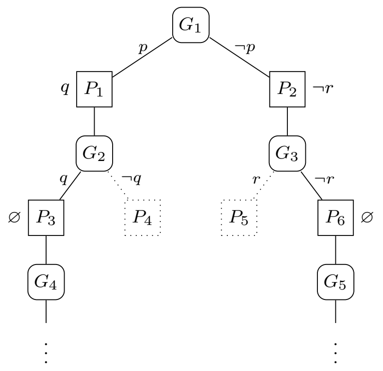
We define enablement-checking variants of FIFO and RR as follows:

– The $\mathsf{FIFO}^{\mathsf{E}}$ approach selects the first enabled intention in its queue, and maintains focus until the intention finishes, blocks, or reaches a point where the current

[3] Where no intention is enabled some non-enabled intention is selected. Progressing this intention will, if possible, result in failure recovery by choosing a different plan for the current, or some ancestor goal. If no alternative plan is available, it will result in the failure of that intention.



(a) Goal-plan tree annotated with coverage measures.



(b) Goal-plan tree annotated with plans' p-effects.

Fig. 2 Two goal-plan trees. Edges between goals and plans are annotated with plans' context conditions.

goal has no applicable plan. When no intentions are enabled, $\mathsf{FIFO}^{\mathsf{E}}$ behaves as FIFO would by selecting (and applying failure recovery to) the intention at the head of the queue.

– The RR^{E} scheduler, like RR, selects intentions cyclically, but skips over any intentions which are not enabled. If no intentions are enabled, it selects an intention (for failure recovery) as per normal RR behaviour. The selected intention is executed for a set number of steps, or until it finishes, blocks or is no longer enabled.

3.2 Low coverage prioritisation

The intuition behind low-coverage prioritisation is that if a given intention is "vulnerable," in that it can only successfully execute in a limited set of world states, then it should be prioritised for execution when an opportunity arises. The vulnerability of intentions is assessed using a refined version of *plan coverage* as introduced for agent-oriented software engineering [22]. Coverage denotes the degree of completeness of the know-how available. The lower the coverage level of an event-goal, the fewer situations there are in which know-how for achieving the goal is available, and the more vulnerable the goal is. As a plan's context condition is a logical formula, the coverage of an event-goal can be calculated by performing model counting [14] over the context conditions of its relevant plans, i.e., by calculating the proportion of world-states in which the goal will have at least one relevant and applicable plan. However, the robustness of an event-goal must take into account any gaps in know-how of subgoals futher down the goal-plan tree. To this end, we apply model counting to define a recursive technique for calculating the coverage of goals, plans and complete intentions. Our *low coverage prioritisation* approach $\mathcal{C}$ is always focused on an intention with the lowest aggregated coverage level over all of the intention's pending event-goals (that is, those event-goals that the intention will eventually need to resolve). Let us now make these intuitions precise.

Consider first an event-goal G that the agent may need to resolve. The question is *how robust is the agent's know-how information for G?* One can answer that question—or at least approximate it—by looking at the number of possible situations in which there are plans available in the agent's plan library to handle the goal in question. In order to tackle various scenarios, including unexpected changes, it is desirable for a BDI plan library to contain, for each goal of concern, a variety of strategies for different potential situations (i.e., contexts). Hence, an aircraft controller may include landing procedures for different weather conditions. However, in many applications, it is not feasible, or even possible, to provide concrete strategies for every possible situation. Like any knowledge, the agent's know-how will be intrinsically incomplete and have "gaps." For instance, consider the goal-plan tree depicted in Figure 2(a) for goal G_1. The goal has two plan rules in the plan library, namely, $P_1 = G_1 : p \leftarrow \delta_1$ and $P_2 = G_1 : \neg p \leftarrow \delta_2$, covering the whole set of possible world states. This means that the plan library will *always* be able to prescribe a strategy for handling G_1 thus increasing G_1's robustness. On the other hand, the plan library has a know-how "gap" for goal G_2 when proposition q is believed false.

Even when a goal has complete "basic" coverage, as it is the case with goal G_1, its robustness can be compromised by that of subgoals lower down in the hierarchy. In our example, plans P_1 and P_2 require the resolution of other (sub)goals. If plan P_2 happens to be applied (because p was false), and when trying to address subgoal G_4 the agent happens to believe that formula $\neg s \wedge \neg t$ is true, then no useful plan is applicable for handling G_4. This could cause the failure of G_4 and, in turn, of G_1.[4] In other words, lack of coverage for subgoals introduces vulnerabilities into the plans that posted them. A more accurate estimation of coverage should then go beyond basic coverage, and potential "gaps" in the sub-tree below a plan need to be propagated up (e.g., gaps in G_4 and G_2 need to be represented in the coverages of P_2 and P_1, respectively)

In addition to the impact of subgoals on the robustness of higher-level goals, multiple plans for a given goal may be applicable for a given world state. Either plan P_4 or P_5 can be used to handle goal G_3 in states where proposition s is believed true. In agent-oriented software engineering terms, this is referred to as *plan overlap*, a desirable property that increases behavior flexibility [22].

Since plans' context conditions are logical formulas, we capture the notion of coverage for formulas, goals, plans, and intentions by resorting to model counting [14]. Given a formula ϕ, its coverage level, denoted $C(\phi)$, is the proportion of models in which ϕ holds true. Technically, $C(\phi) = \frac{\#(\phi \wedge \Theta)}{\#\Theta}$, where $\#\phi$ is the number of models for ϕ and Θ is a boolean formula encoding the available consistency constraints of the domain (e.g., elevators can only be at one floor at a time). If no constraints are available, we just take $\Theta = \texttt{true}$ (in which case $\#\Theta = 2^{|L|}$ where L is the set of all belief propositions of the domain).[5]

[4] Remember that BDI plan libraries are generally developed in a modular, incremental, and independent manner, so plans for G_1 and G_4 may have been developed separately. Hence the context condition of plan P_2 may not account for the incomplete coverage of plans P_6–P_8.

[5] Of course, the more domain constraints encoded, the more accurate coverage estimations will be obtained. In practice, we expect many domain constraints are readily available at design time.

Then, we define the *coverage for goals and plans* as follows:[6]

$$C(G) = \sum_{\{X \mid X \subseteq \Delta_G\}} C\Big(\bigwedge_{\{\phi \mid (\phi, P) \in X\}} \phi \wedge \bigwedge_{\{\phi \mid (\phi, P) \in \Delta_G \setminus X\}} \neg\phi \Big) \times \max\{C(P) \mid (\phi, P) \in X\};$$

$$C(P) = \prod_{\{G \mid\ !G \in P\}} C(G),$$

where $\Delta_G = \{(\phi, P) \mid\ G : \phi \leftarrow P \in \Pi\}$ is the set of all relevant plans for goal G. In words, the coverage of a goal G is obtained by summing over the coverage level of each possible overlap "space" of its relevant plans. Concretely, for each possible subset of plans $X \subseteq \Delta_G$, we consider the proportion of the state-space in which all plans in X, and no others, are applicable (first product term in $C(G)$'s summation). In order to take into account coverage gaps lower down in the goal-plan tree, this value is in turn multiplied by the coverage of the most robust plan in X (second term in $C(G)$'s summation). This yields a coverage value for exactly the area of the state space in which all plans in X overlap. The overall summation over all possible subsets X of Δ_G makes sure all possible overlap spaces are taken into account. Note that this coverage definition accounts trivially for the cases where there is no plan overlap—as the case of goal G_1 in Figure 2—when set X in the summation is a singleton.

In turn, since we have assumed linear plan bodies only, the coverage $C(P)$ of a plan P itself is just the product of the coverage of all of its subgoals ($!G \in P$ means goal G is posted by plan body P).

Consider for example the coverage for goal G_3 in Figure 2(a). Its two relevant plans, P_4 and P_5, do not cover the entire state space, and there is some overlap between them. Since the set of plans relevant to G_3 is $\Delta_{G_3} = \{(\phi_{P_4}, P_4), (\phi_{P_5}, P_5)\}$, where $\phi_{P_4} = r \vee s$ and $\phi_{P_5} = s \vee t$, the coverage for the goal is given by:

$$\begin{aligned} C(G_3) = \;& C(\phi_{P_4} \wedge \phi_{P_5}) \times \max(\{C(P_4), C(P_5)\}) + \\ & C(\phi_{P_4} \wedge \neg\phi_{P_5}) \times \max(\{C(P_4)\}) + \\ & C(\neg\phi_{P_4} \wedge \phi_{P_5}) \times \max(\{C(P_5)\}) + \\ & C(\neg\phi_{P_4} \wedge \neg\phi_{P_5}) \times \max(\{\}). \end{aligned}$$

Each term in the sum corresponds to one of the four possible subsets X of Δ_{G_3}. In each term, the maximum coverage among the applicable plans is taken so as to account for the coverage gaps deeper within the goal-plan tree. The first term corresponds to the domain space in which both plans P_4 and P_5 are applicable, whereas the last term accounts for the space where no plan is available for the goal in question. In this case, because neither P_4 nor P_5 post any subgoals, we have that $C(P_4) = C(P_5) = 1$. So, assuming no domain constraints (i.e., $\Theta = \texttt{true}$) the above formula reduces to:

$$\begin{aligned} C(G_3) &= \frac{\#((r \vee t) \wedge (s \vee t))}{2^3} + \frac{\#(r \wedge \neg s \wedge \neg t)}{2^3} + \frac{\#(t \wedge \neg r \wedge \neg s)}{2^3} \\ &= 5/8 + 1/8 + 1/8 + 0 = 0.625 + 0.125 + 0.125 = 0.875. \end{aligned}$$

[6] We assume the product, conjunction, and maximum over an empty set of elements is equal to 1, `false`, and 0, respectively. Also, for legibility, we shall sometime abuse notation and treat lists or sequences (e.g., a plan body program P or intention I) as sets.

Observe that, when there is no overlap among a set of plans, the conjunction of their context conditions yield an unsatisfiable formula with coverage equal to 0.

We now have all the machinery to define the robustness level of a current active intention and our coverage-based intention scheduling scheme. Recall that given an intention of the form $I = [(G_n : \delta_n)(G_{n-1} : \delta_{n-1}) \cdots (G_1 : \delta_1)]$, each δ_i is a partly executed program that ought to be eventually fully executed for I to complete. So, we generalize the notion of coverage to intentions as follows:

$$C(I) = \prod_{\{G \mid \; (G':\delta) \in I, G \in \delta\}} C(G).$$

Thus the coverage of an intention is the product of the coverage of the goals yet to be handled within that intention: that is the goals within the remaining part of each selected, and possibly partially completed, plan (δ_i).

Finally, the low-coverage prioritisation intention scheme is as follows:

$$\mathcal{C}(\mathcal{I}) = \operatorname*{argmin}_{I \in \mathcal{I} \text{ and } I \text{ is enabled}} C(I).$$

That is, at every execution cycle, $\mathcal{C}$ selects an intention with the lowest coverage among those in the agent's intention base that are enabled.

3.3 Evaluating Intention Selection Techniques

A key constraint on any intention selection technique is that it must be extremely efficient, so as not to compromise the soft real-time characteristics of BDI systems. Our techniques imply a runtime overhead which is linear in the number of intentions: in the worst case, every intention may need to be checked for enablement. Enablement checking is the process of assessing whether there is an applicable plan for a goal pending to be resolved. Given that this process already exists and is highly optimised in BDI systems such as Jack, and that typically only a few extra intentions would need to be checked before finding one that was enabled, we do not expect the additional runtime cost to be problematic.

Because coverage levels do not depend on the state of the environment, but only on the static goal-plan tree structures, the coverage of goals and plans can be computed offline and stored (e.g., as annotations in the goals and plans themselves) for $\mathcal{C}$ to refer to at runtime.[7] In terms of offline computation, given that we assume that plan libraries induce finite, non-recursive goal-plan tree hierarchies, coverage of goals and plans can be calculated by simply scanning these structures from the bottom up and processing each goal and plan once. The main effort lies in computing the coverage of all possible subsets of relevant plans for each goal in the plan library (i.e., the first product term in the definition of $C(G)$), as for each such subset a model counting task must be performed. Thus, the efficiency of this process is dependent on the efficiency of the model-counter used, and the difficulty of the specific model counting

[7] Observe that it is also possible to compute (and store) offline the coverage of every partially executed plan, so that even $C(I)$, for any partially executed intention I, can be reduced to a table lookup.

tasks encountered. It is known that while many model counting tasks can be solved very efficiently, there are some tasks which are very hard, such as instances with large numbers of variables and a clause to variable ratio around 1.5 [14]. While not the focus of this paper, we did some preliminary tests with exact counter Cachet [27] and found that for some arbitrarily chosen problems with up to 50 variables, and a clause/variable ratio of 1.5, the time taken was about 0.5 seconds. Increasing to 100 variables became problematic at about 3 minutes. Arbitrary 100 variable problems with a clause/variable ratio of 4.3 were solved in about 0.05 seconds. In our experience (which includes working with commercial agent system development), real BDI programs would be highly unlikely to have more than 50 variables across the set of plans for a particular goal, and they also typically have a very limited number of relevant plan types for each goal (less than 5). Based on our preliminary measures, then, if we assume an average of 5 plan types per goal, with 50 variables and the most difficult clause/variable ratio of 1.5, we would predict an offline compilation time of approximately 4.5 hours. By contrast, a similar problem where all clause/variable ratios were favorable could be expected to take around 10 minutes. While this clearly requires more substantial investigation with real BDI programs, we believe this provides sufficient basis to consider the approach feasible. If necessary, one can also resort to approximate model counting techniques, which have been shown to be highly accurate while scaling much better than exact model counters [36].

Besides the necessary requirement of efficiency, one wants to analyse how many of its (top level) goals an agent manages to achieve, that is, how many intentions run to completion. To that end, we measure the agent's *success rate* as the ratio between the number of successfully completed intentions and the total number attempted. In addition, for some of our experiments (Section 5), we shall also analyse the following three additional, subsidiary measures.

Failure recovery use This is the the number of failure recoveries triggered by the agent, divided by the number of steps in an agent execution. A failure recovery amounts to the agent starting an alternative strategy for a given goal, when the current strategy (i.e., plan) fails for some reason. This is a powerful feature of BDI systems, as it makes them robust in dynamic environments. However, excessive reliance on the failure recovery mechanism is likely to result in wasted resources due to the execution of unsuccessful paths. Consequently, all else being equal, one would prefer a low reliance on failure recovery. Since there could only be at most one failure recovery per agent execution step, this measure ranges between 0 (no failure recovery used) to 1 (every step is a recovery step). The measure was not used in the synthetic programs (Section 4), as they provide little opportunity for meaningful failure recovery.

Efficiency This is the number of moves taken by the program to solve the Tower of Hanoi task on each execution (Section 5). For simplicity, and because we do not assume any extra domain information beyond a standard BDI program, we assumed a unit cost model. Of course, in the context of a domain-specific non-unit cost model, it would be desirable to incorporate cost as part of the intention selection model (see Section 7).

Fairness This aims to measure how "evenly" the agent is sharing its focus of attention across its active intentions. Managing multiple goals—different focuses of attention [6]—is an important aspect of practical reasoning. In general, a rational agent system should strive to progress all of its goals "equally." If an agent is attending to many customers, we do not want some customers waiting for long periods, while others use all of the agent's resources and are expedited.

We base our fairness measure on that proposed by Jain et al. [20] to determine whether a resource (in our case, execution time steps) is being allocated fairly to a set of users (in our case, intentions). For allocation data $\mathbf{x} = x_1, \ldots, x_n$, where $x_i \geq 0$ represents the amount of the resource (e.g., time) received by user i, for each $i \in \{1, \ldots, n\}$, Jain et al.'s fairness index is calculated as follows:

$$\mathit{fairIdx}(\mathbf{x}) = \frac{(\frac{1}{n}\sum_{i=1}^{n} x_i)^2}{(\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2})^2} = \frac{(\sum_{i=1}^{n} x_i)^2}{n \times \sum_{i=1}^{n} x_i^2}.$$

That is, the fairness index amounts to the ratio between the square of the arithmetic mean and the square of the quadratic mean. By the power mean inequality rule and the fact that $x_i \geq 0$, for all $i \in \{1, \ldots n\}$, it is the case that $\mathit{fairIdx}(\mathbf{x}) \in [0, 1]$. In particular, $\mathit{fairIdx}(\mathbf{x}) = 0$ when $x_i = 0$, for all $i \in \{1, \ldots n\}$ (i.e., no allocation of resources), whereas $\mathit{fairIdx}(\mathbf{x}) = 1$ when all values of x_i are equal (i.e., fully fair allocation). If the resources are equally distributed among k users, and the remaining $n - k$ receive nothing, then $\mathit{fairIdx}(\mathbf{x}) = \frac{k}{n}$. It turns out that the measure is independent of the population size and the scale of the values of $\mathbf{x}$ (i.e., multiplying all values x_i with some constant does not effect the final index), and is continuous (i.e., a small change in any value in $\mathbf{x}$ results in a small change in the index). Importantly for our setting, Jain et al.'s fairness index is suitable for cases where a fair allocation of resources does not necessarily imply an equal allocation. In such cases, one simply defines x_i as the ratio of the amount of the resource allocated to user i to how much allocation it "should" have received under a completely fair scheme.

Let us apply the above fairness measurement to intention scheduling. First of all, one needs to be able to refer to an "intention" regardless of its current form as it evolves over agent execution. We do so by assuming that each intention structure I is assigned a unique, fresh, identifier $\mathit{id}(I)$ when added to the intention base, which is kept throughout its execution, i.e, $\mathit{id}(I_i) = \mathit{id}(I_j)$ for all $i, j \in \{1, \ldots, n\}$ whenever $I_1, \ldots, I_n$, with $I_1 = I$, is the *complete* execution of intention I. Next, we need to take into account that, in many cases, simply allocating equal amounts of time steps to each intention will not produce a fair distribution of resources, as not all intentions will have the same duration. So, consider sequence $\boldsymbol{\mathcal{I}} = \mathcal{I}_1, \ldots, \mathcal{I}_\ell$ to be the evolution of an agent's intention base for $\ell \geq 1$ execution steps. We define the *amount of time a completely fair agent should have spent* on a given intention k as follows (again, while intention base evolution $\boldsymbol{\mathcal{I}}$ and intention I are sequences, we treat them as sets):

$$E(k) = \sum_{\{\mathcal{I}_i \mid \mathcal{I}_i \in \boldsymbol{\mathcal{I}},\ I \in \mathcal{I}_i,\ \mathit{id}(I) = k\}} \frac{1}{|\mathcal{I}_i|}.$$

The important aspect of this definition is that it takes into account how many intentions were competing for attention at *each* step when intention k was active, because the number of intentions in the agent's intention base may change over time, as new intentions are added and finished intentions are removed. So, if at a given time step an agent has n running intentions, then they should each receive $\frac{1}{n}$ of the agent's attention. This can be extended to a complete execution sequence by calculating $E(k)$ as the sum of $\frac{1}{|\mathcal{I}_i|}$ for each step i that k was present in the agent's intention base.

For example, consider an agent with an initial intention base $\mathcal{I}_1 = \{I_1, I_2\}$. Suppose that both intentions require exactly three execution steps to complete, and hence, the whole intention base can be finished in a total of six execution cycles. This implies an intention base run of the form $\boldsymbol{\mathcal{I}} = \mathcal{I}_1, \dots, \mathcal{I}_7$, where $\mathcal{I}_7 = \emptyset$. Under a round robin scheme, the intentions will be selected for execution in the order $I_1, I_2, I_1, I_2, I_1, I_2$. Note that by step six, intention I_1 has been fully executed, and therefore intention base $\mathcal{I}_6$ only contains (what remains of) intention I_2. That is, $|\mathcal{I}_6| = 1$ and $|\mathcal{I}_i| = 2$, for all $1 \leq i \leq 5$. So, intention I_1 was present in intention bases $\mathcal{I}_1, \dots, \mathcal{I}_5$ only, and thus the time that it *should* have received from the agent is $E(I_1) = \frac{1}{2}+\frac{1}{2}+\frac{1}{2}+\frac{1}{2}+\frac{1}{2} = 2.5$. I_2 was present throughout the execution, however at step six it was the only intention present. Hence $E(I_2) = \frac{1}{2}+\frac{1}{2}+\frac{1}{2}+\frac{1}{2}+\frac{1}{2}+\frac{1}{1} = 3.5$.

Next, using this "expected" measure $E(k)$, an intention's actual allocation—the degree to which it was starved or favoured by the agent—is the ratio of the observed $O(k)$ and expected $E(k)$ time spent by the agent on the intention, that is, $\mathit{alloc}(k) = O(k)/E(k)$. For example, if $\mathit{alloc}(k_1) = 2$ and $\mathit{alloc}(k_2) = .25$, then intention k_1 received twice its fair share of attention, whereas intention k_2 received one quarter.

Finally, consider the finite evolution of an agent's intention base $\boldsymbol{\mathcal{I}} = \mathcal{I}_1, \dots, \mathcal{I}_\ell$, and let $\Delta_{\boldsymbol{\mathcal{I}}} = \{k \mid I \in \bigcup_{i=1}^{|\boldsymbol{\mathcal{I}}|} \mathcal{I}_i,\ k = \mathit{id}(I)\}$ be the set of all intention identifiers along $\boldsymbol{\mathcal{I}}$. We define the fairness index for such an execution as follows:

$$\mathit{fairIdx}(\boldsymbol{\mathcal{I}}) = \frac{(\sum_{k \in \Delta_{\boldsymbol{\mathcal{I}}}} \mathit{alloc}(k))^2}{|\Delta_{\boldsymbol{\mathcal{I}}}| \times \sum_{k \in \Delta_{\boldsymbol{\mathcal{I}}}} \mathit{alloc}(k)^2}.$$

Again, this index will range between $\frac{1}{|\Delta_{\boldsymbol{\mathcal{I}}}|}$ (least fair) and 1 (most fair), is independent of the number of intentions and the scale of the time allocations, and does not require equal intention scheduling to achieve full fairness.

Let us return to the two-intention execution example above. As both intentions were executed for three steps, $O(I_1) = O(I_2) = 3$. Using the formulas above, the intentions' allocations can be calculated: $\mathit{alloc}(I_1) = \frac{3}{2.5} = 1.2$, and $\mathit{alloc}(I_2) = \frac{3}{3.5} = 0.86$. These two allocations give a fairness index $\mathit{fairIdx}(\boldsymbol{\mathcal{I}}) = 0.97$. Interestingly, this shows that a round robin scheme does not result in a perfectly fair allocation. By virtue of being selected first, I_1 has been slightly favoured by the agent, while I_2 has been made to wait. However, in general, the longer the round robin scheme runs, the less effect this initial imbalance has on the overall fairness index.

Consider, lastly, an alternative run $\boldsymbol{\mathcal{I}}' = \mathcal{I}_1, \mathcal{I}'_2, \mathcal{I}'_3, \mathcal{I}'_4$ in which the agent has only run intention I_1, and ignored intention I_2. The execution sequence is thus I_1, I_1, I_1, and so $|\mathcal{I}_1| = |\mathcal{I}'_2| = |\mathcal{I}'_3| = 2$ and $\mathcal{I}'_4 = \{I_2\}$. Here, $E(I_1) = \frac{1}{2} + \frac{1}{2} + \frac{1}{2} = 1.5$ and $E(I_2) = \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{1} = 2.5$. Since $O(I_1) = 3$ and $O(E_2) = 0$, we obtain $\mathit{alloc}(I_1) = 2$ and $\mathit{alloc}(I_2) = 0$, and a final fairness index $\mathit{fairIdx}(\boldsymbol{\mathcal{I}}') = 0.5$.

In order to evaluate the performance of the algorithms discussed in Sections 3.1 and 3.2, we ran experiments on synthetic domains, with many automatically generated programs, as well as a single meaningful program. More concretely, we first evaluated both the enablement checking and low-coverage intention selection approaches on a set of synthetic domains in terms of the agent's success rate. Based on the results obtained here, we further evaluated the enablement checking approaches on a real program, analysing also their failure recovery, efficiency, and fairness. We describe the details of these evaluations in the following two sections.

4 Experimental Evaluation Using Synthetic Domains

We have first performed a large number of experiments using automatically generated goal-plan trees (i.e., BDI programs) with different coverage levels, which were executed in environments with varying levels of dynamism. The use of batch-generated synthetic goal-plan trees provides a number of advantages: it is possible to explore large numbers of programs, with tight control over their characteristics, although the simplifications required necessarily introduce some limitations. We first describe the structure of those synthetic programs and how different levels of coverage were achieved, and then discuss the results of our experimentation.

4.1 Program/testbed structure

The goal-plan trees automatically generated for our experiment resemble a binary decision tree, where each goal is handled by two plans with complementary atomic propositional context conditions. Each plan may post at most one subgoal—leaf nodes post no goals. Restricting to such trees simplifies the automatic generation and control of such tree structures. Using atomic context conditions with no overlap enables us to calculate the coverage measure directly, without using a model counter as proposed in [33].[8]

Coverage gaps via preparatory effects The most important aspect of our synthetic programs is the nature and form of the coverage "gaps." Agent-oriented design methodologies, such as Prometheus [22], recommend that agent program developers carefully consider any incomplete goal coverage in their design. Nevertheless, for a variety of reasons, few programs actually have full coverage at all points. In the authors' experience, one typical reason for lack of complete coverage in agent programs is that developers write code so that the agent sets up conditions required by a subsequent subgoal. These "conditions preparing for later steps" are known as *preparatory effects* or just *p-effects* [31].[9] The assumption is that, once in place, those conditions will not be threatened, and therefore there is no need to provide plans (for the subgoal in question) for situations in which those conditions do not hold, hence creating a "coverage gap."

[8] The simplification of at most one subgoal posting per plan and no overlap between plans' context conditions is not trivial, but we believe is justifiable to enable us to gain a well-structured understanding of the intention selection approaches.

[9] Work has been done to recognise these p-effects and ensure that the agent does not itself undo them prematurely [31].

For example, in the goal-tree structure shown in Figure 2(b), there is no plan for subgoal G_2 in situations where proposition q is false (i.e., no plan for G_2 with context condition $\neg q$). However, plan P_1 makes q true, thus "preparing" the conditions for plan P_3 to be suitable for achieving G_2. The *setup distance* is the number of subgoals between the setting of the p-effect and its use in the context condition of a subsequent plan. We have used a setup distance of one in all reported experiments.[10]

In the absence of any environmental changes *and* any interleaving of the agent's intentions, the lack of coverage due to reliance on p-effects will have no adverse effect, as plans will be "chained" as the developer assumed. Under such (strong) assumptions, plan P_3 will be applicable when G_2 is ready to execute, and the agent will experience no problems from G_2's lack of coverage. However, agents are intended to operate in dynamic environments and should be able to interleave intentions.

In order to model coverage gaps based on p-effects, the basic binary structure is modified to create a tree with less than full coverage by removing one of the plans which handle a goal (e.g., removing plans P_4 and P_5 in Figure 2(b)), and adding an appropriate p-effect into one of the goal's ancestors (e.g., adding q and $\neg r$ as p-effects of P_1 and P_2, respectively).

Size of coverage gaps Due to the binary structure of the goal-plan trees, removing a plan using the "pruning" technique described above leaves the corresponding goal without an applicable plan for half of the state space.

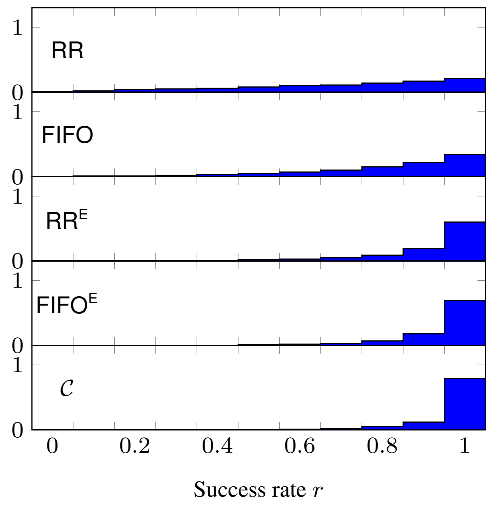
However, for our experimentation, we want to create coverage gaps of any arbitrary size. We do so by adjusting the distribution of propositions. More concretely, we relax the implicit assumption that propositions have uniform distributions, and instead allow a proposition p to be true under a given probability $\alpha_p \in [0, 1]$. This effectively means that the proportion of world states in which proposition p is true is no longer 0.5 (or 50%), but instead equal to α_p.

For example, consider again goal G_2 in Figure 2(b). A coverage gap under G_2 has been produced by removing plan P_4 and adding q as a p-effect to plan P_1. The area of the state-space in which the remaining plan P_3 is applicable, and hence the size of the coverage gap, can be set by adjusting the truth probability of proposition q. If $\alpha_q = 0.9$, then P_3will be applicable 90% of the time, and will effectively cover 90% of the state space.

Agent's beliefs and dynamic environments As is customary, our agents track (the state of) the environment using a finite set of *belief propositions*. In BDI systems, such propositions are used in agents' plans, and in particular, in plans' context conditions to support adequate means-end analysis. In our experiments, an *environment variable* is a tuple $\langle p, \alpha, b\rangle$ comprised of a belief proposition p, its sampling probability α, and a *boolean* value $b \in \{\mathit{true}, \mathit{false}\}$ representing p's current state.

As we assume such systems operate in dynamic environments, we expect those propositions to sometimes change unexpectedly, that is, without the intervention of the agent. How frequent those changes are is determined by the so-called *dynamism*

[10] We found no interesting experimental results when varying this value, other than simply confirming the expected fact that the greater the setup distance, the more likely it is that this dependency will be broken, causing the intentions to fail or suspend more frequently.



(a) Histogram showing the distribution of the success rate. The x-axis indicates the success rate r, divided into bins of width 0.1. The y-axis shows the proportion of tests which fall into each bin, from all tests (1) to none (0).

Scheduler	μ_r	σ_r
FIFO	0.825	0.204
RR	0.728	0.256
FIFO^E	0.951	0.101
RR^E	0.909	0.144
$\mathcal{C}$	0.963	0.079

(b) The mean (μ) and standard deviation (σ) of the success rate r for the standard algorithms (FIFO and RR), their enablement-checking variants (FIFO^E and RR^E), and low-coverage prioritisation ($\mathcal{C}$). Values were calculated by averaging across all tests, i.e., all levels of coverage and environmental dynamism.

Fig. 3 The success rates of all algorithms, averaged over all tests.

rate $d \in [0, 1)$ of the environment, where $d = 0$ represents a fully static environment and $d = 1$ stands for highly dynamic one.

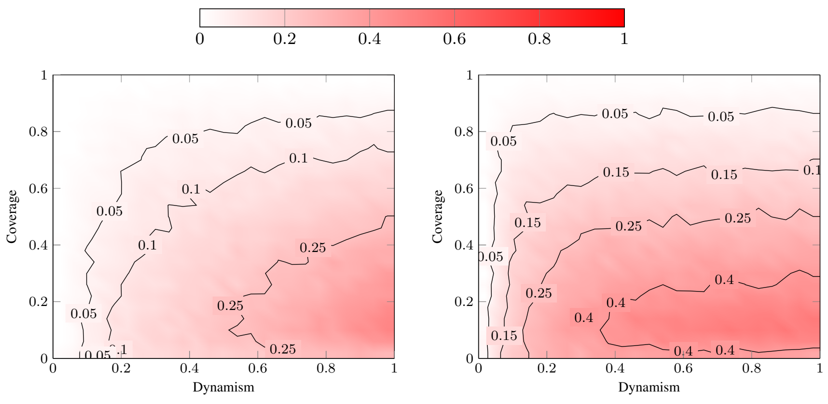
At every reasoning step, random changes are applied to the environment based on the dynamism rate. More concretely, any given environment variable $\langle p, \alpha, b\rangle$ is re-sampled with probability d; if p is re-sampled, then its current state b is set to *true* with probability α (and *false* with probability $1 - \alpha$).

4.2 Evaluating success with varying coverage and dynamism

We evaluated the intention selection mechanisms $\mathcal{C}$, FIFO^E, and RR^E with respect to their ability to successfully complete intentions in situations with different coverage and dynamism, as compared to the standard FIFO and RR mechanisms.

For each test run, an average coverage level $c \in [0.01, 0.99)$ and a dynamism rate $d \in [0, 1)$ were first randomly selected. Then, an agent was built with 10 randomly generated top-level goals which can be decomposed into goal-plan tree structures of the type described above, and with a coverage of c. Finally, the agent is run in an environment with a dynamism rate d and an initial state sampled according to the probability, α_i, of each environment variable, p_i being true. (For further details see Appendix A). We ran $100{,}000$ such tests, to get a good representation across all coverage levels and dynamism rates. In all graphs below, the data has been put into buckets of size 0.05.

Table 3(b) summarizes the results for all algorithms, showing the mean and standard deviation of their success rates across all coverage levels and dynamism rates.



(a) $\mathsf{FIFO}^{\mathsf{E}} - \mathsf{FIFO}$ (b) $\mathsf{RR}^{\mathsf{E}} - \mathsf{RR}$

Fig. 4 The change in success rate due to incorporating enablement checking into FIFO and RR. A difference of 0 (white) indicates no effect and a difference of 1 (red) indicates an improvement of 100pp. Contour lines indicate the point at which the difference in success rate crosses a specified threshold.

The normalised histogram in Figure 3(a) depicts the patterns of success for the different algorithms, averaged over all coverage and dynamism rates. As can be seen, the most successful is $\mathcal{C}$, with all the enablement checking options showing a similar pattern of high levels of 100% success.

Figures 4 and 5 compare the success rates of different algorithms under different levels of coverage and rates of dynamism. The difference is measured in *percentage points* (pp). For example, if scheme A has a success rate of 0.3 (i.e., 30% of intentions complete) and scheme B has a success rate of 0.4 (i.e., 40%), we then say that B improves on A by 0.1 or 10pp. Figure 4 compares standard FIFO and RR with their enablement checking counterparts, giving an overview of the value of this technique in achieving successful execution of all intentions. Figure 5 compares $\mathcal{C}$ with both standard FIFO and RR, as well as with their enablement checking counterparts. The former provides information on the value of low coverage prioritisation as compared to standard techniques FIFO and RR, while the latter shows the added benefit of low coverage prioritisation over enablement checking alone, as enablement checking is necessarily incorporated into the low coverage prioritisation approach.

Key results The experiments show conclusively that enablement checking is of substantial value, while the additional aspect of low coverage prioritisation can provide further benefit, especially in adverse situations. For all comparisons, a significance threshold of 0.001 was used. More precisely:

1. Enablement checking in intention selection is never detrimental to the success rate, and provides an increase of up to 47.9pp to FIFO and 52.8pp to RR. By prioritising vulnerable intentions, low coverage prioritisation scheme $\mathcal{C}$ increases this to 64.9pp and 67.8pp.
2. For all but the most static environments and robust programs, $\mathcal{C}$ is the most effective intention selection mechanism.

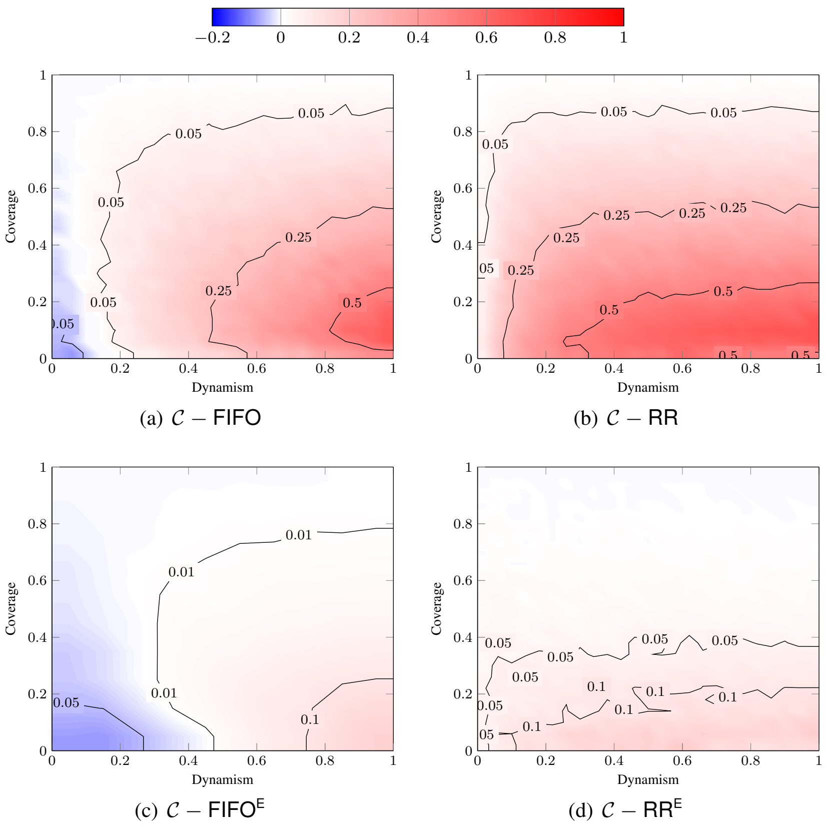
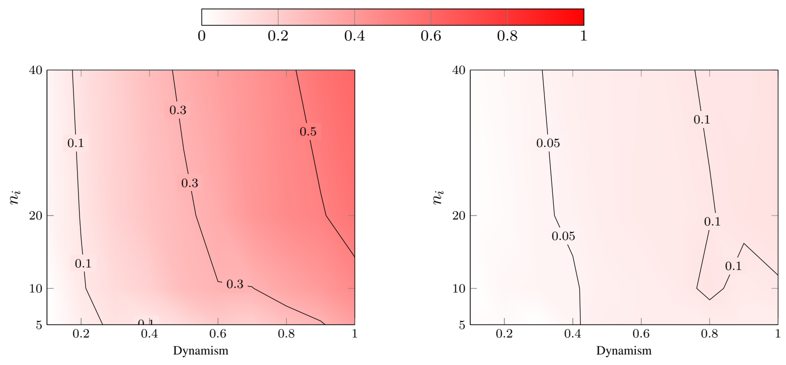


Fig. 5 A comparison of the success rate of $\mathcal{C}$ with FIFO$^{\mathsf{E}}$, RR$^{\mathsf{E}}$, FIFO and RR.

3. Enablement checking, which is built in to the low coverage prioritisation scheme, is responsible for the largest part of $\mathcal{C}$'s performance. The extra benefit from $\mathcal{C}$ is greatest when running fragile programs in highly dynamic environments, reaching a maximum additional benefit of 17.9pp over FIFO$^{\mathsf{E}}$ and 20.6pp over RR$^{\mathsf{E}}$.
4. Low coverage prioritisation scheme $\mathcal{C}$ turns out to be slightly detrimental as compared to FIFO$^{\mathsf{E}}$ or even FIFO when the environment is relatively static. This is because the focus switching caused by low coverage prioritisation, also creates greater (negative) interaction between intentions, for which there are few opportunities for the (static) environment to mitigate.

4.3 Evaluating success with many interacting intentions

In the previous experiments we used 10 intentions per run, on the basis that, in the authors' experience, this is an approximate upper bound on the number of top-level goals *of different types* that an agent would be likely to be managing concurrently. However, in some applications agents may well have larger numbers of *instances* of



(a) $\mathsf{FIFO}^{\mathsf{E}} - \mathsf{FIFO}$, $0.2 \leq c < 0.3$ (b) $\mathsf{FIFO}^{\mathsf{E}} - \mathsf{FIFO}$, $0.8 \leq c < 0.9$

Fig. 6 The effect of enablement checking on FIFO for two coverage levels (c) under different rates of environmental dynamism and numbers of interacting intentions (n_i).

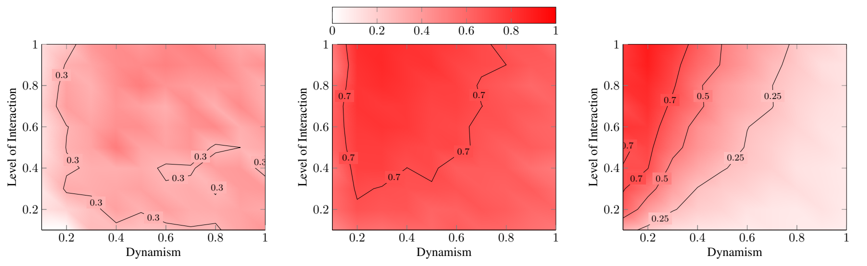
a single type of task, such as in a warehouse or factory. Such intentions would likely interact with each other (as they are instances of the same goal type, likely competing for the same resources).

To evaluate this we constructed goal-plan trees (i.e., programs) with an identical structure, but with potentially different variables in context conditions, to represent different instances of the same top level goal/intention. Negative interaction between intentions was achieved by having actions that set p-effects in one tree negate the p-effects of other trees, with the number of such occurrences controlling the amount of interaction/interference. Test runs varied the number of intentions between 5 and 40, as well as the amount of interaction, environmental dynamism, and average coverage of programs. We ran $40{,}000$ tests on each algorithm, recording for each the proportion of successfully completed intentions.

Figure 6 summarizes the advantages enablement checking provides when applied to FIFO. Consistent with previous results, the greatest benefit is seen in low coverage programs ($0.2 \leq c < 0.3$) running in highly dynamic settings (c.f. Figure 6(a)). This benefit is somewhat increased as the number of intentions increases, as the agent is more likely to have at least one enabled intention. The level of interaction between intentions is unimportant, as there is little switching between intentions under FIFO, thus creating little opportunity for interference.

Figure 7 shows the somewhat more complex situation with regard to the effect of enablement checking on RR. Under RR and RR^{E}, there are complex interactions between environmental dynamism, number of intentions, level of interaction and the cyclic nature of the RR approach. As before, the advantage is greatest when intentions have low coverage ($c < 0.3$), though it is now accentuated when there are larger numbers of intentions (see Figures 7(a) and 7(b)).

Figure 7(c) depicts the situation with relatively high coverage (i.e., $c \geq 0.7$). Here, we see that the greatest benefit of enablement checking is when running highly inter-related intentions in relatively static environments. This is understandable, as there are negative effects of interaction between intentions when the focus between them switches. In a relatively static environment, there is reduced opportunity for the



(a) $\mathsf{RR}^{\mathsf{E}} - \mathsf{RR}, n_i = 5, c < 0.3$ (b) $\mathsf{RR}^{\mathsf{E}} - \mathsf{RR}, n_i = 40, c < 0.3$ (c) $\mathsf{RR}^{\mathsf{E}} - \mathsf{RR}, n_i = 40, c \geq 0.7$

Fig. 7 The effect of enablement checking on RR scheduling with multiple interacting intentions.

environment to serendipitously undo the negative effects of interaction. This advantage is somewhat more pronounced with larger numbers of intentions.

We also looked at the additional effect of selection scheme $\mathcal{C}$ over $\mathsf{FIFO}^{\mathsf{E}}$. As one would expect, $\mathcal{C}$'s low coverage prioritisation yields most added benefit in those situations where the interaction between intentions is most destructive, that is, when there is low dynamism and high interaction. Specifically, the extra advantage of $\mathcal{C}$ ranges from 2.8pp in higher coverage situations to a high of 11.7pp in relatively low coverage scenarios. The number of intentions lessens the advantage provided by $\mathcal{C}$, as enablement checking tends to have already provided all available gains.

Key results Analysis of the results in the context of interacting intentions shows that:

1. Enablement checking is never detrimental to the success rate achieved by either FIFO or RR.
2. Enablement checking provides a greater advantage to both FIFO and RR as the number of intentions increases. When running 5 low coverage ($c < 0.3$) intentions in a highly dynamic environment ($d = 1$), $\mathsf{FIFO}^{\mathsf{E}}$ has an advantage of up to 37pp over FIFO, increasing to 60pp with 40 intentions. When running 5 low coverage ($c < 0.3$) intentions, RR^{E} provides a benefit of 51pp over RR, increasing to 85pp as the number of intentions increases to 40.
3. Enablement checking provides the most benefit for RR in those situations where interference is most destructive. For example, when running large numbers (40) of highly interactive, high coverage ($c \geq 0.7$) intentions in low dynamism environments ($d < 0.2$), RR^{E} improves on RR by 89pp.
4. Low coverage prioritisation provides a further gain when running low coverage programs, ranging from an increase of 2.8pp to 11.7pp as the coverage decreases from 1 to 0.

5 Tower of Hanoi

Evaluating the intention selection algorithms using automatically generated synthetic programs is valuable due to the possibility of running tests on many different programs. However, these programs are, by necessity, simplified, and there is a danger that some aspect of the simplification may skew the results. For example, the lack of overlap in the plans results in little opportunity for failure recovery, potentially skewing the results in favour of the coverage or enablement checking approaches. Consequently, we have also evaluated the enablement checking intention selection algorithms on one genuine hand-coded program for a robot agent playing an extended version of the well-known Tower of Hanoi game, where the goal is to stack discs of decreasing size onto a single pin.[11]

5.1 Program description

We have suitably modified and extended the Tower of Hanoi problem to capture the following key characteristics of our agent programs:

1. There must be multiple concurrent top-level goals that the agent is trying to achieve, and hence multiple intentions from which the agent must select at every step.
2. There should be opportunities for failure recovery in the event of failure, as this is a key robustness feature of BDI systems. Lack of such opportunities in the synthetic programs places standard intention selection mechanisms at a disadvantage.
3. There must be some unexpected and uncontrollable changes in the environment that could affect the execution of the agent's programs, as such dynamic domains are where BDI agents are most effective (and where failure is likely to occur).
4. There must be some subgoals with partial coverage, that is, there must be some situations in which there will be no applicable plan for a subgoal. Indeed, if there is always an applicable plan, there is less motivation to engage in smarter ways of doing intention selection—there will always be some way to progress a goal.

The standard Tower of Hanoi puzzle consists of three pins, A, B and C, and n differently sized discs, $d_1, d_2, \ldots, d_n$. The discs are numbered according to their sizes, with d_1 being the smallest and d_n the largest. In the game's initial state, all of the discs are placed on pin A in order of size, i.e., with disc d_n at the bottom of the stack and d_1 at the top. The purpose of the game is to move the stack of discs from A to C, while observing the rules that *(i)* only one disc may be moved at a time; *(ii)* a disc may not be placed on top of a smaller disc; and *(iii)* discs may only be moved from the top of a pin.

There is a well-understood recursive optimal solution to this problem, for which a BDI agent program exists as part of the Jack distribution [7]. To address the above requirements, we extended the problem—and its recursive solution—as follows:

1. The agent is programmed to solve, concurrently, multiple instances of a standard 3-pin Tower of Hanoi (ten instances in our experiments). In addition to the standard rules described above, there are two additional constraints placed upon the agent: *(i)* a disc cannot be moved between instances; and *(ii)* in any given

[11] We focused only on enablement checking because, as seen from the experiments reported in Section 4, it is after all what yields most benefits. Moreover, the program structures to be used include recursive subgoal postings, which cannot be, at this stage, handled by the low coverage prioritisation mechanism.

time step, only a single disc from a single tower instance can be moved. Having multiple tower instances ensures that the agent is pursuing multiple top-level goals—one per tower—and hence has multiple intentions from which to select.
2. The 3 pins for each tower instance are arranged in a circle. At the centre of each is a motorised robotic arm which can used by the agent to move discs between pins on that instance only. However, the motors are subject to random deteriorations and repair, and as an arm's motor deteriorates, restrictions are placed on the moves that are available to it. At the first level of deterioration, the arm becomes too weak to move the largest disc in a "clockwise" manner, i.e., from pin A to B, B to C or C to A. Further deteriorations result in restrictions on both clockwise and anti-clockwise movements of increasingly smaller discs. The restrictions remain until the arm is (randomly) repaired. This motor deterioration provides the required dynamism in the environment, which can cause chosen plans to fail during execution. For simplicity, the rate of deterioration and repair is constant across all motors in each test run.
3. In addition to the standard recursive solution, which may now sometimes not be suitable due to unexpected environmental changes in the state of the motors, we introduce an alternative BDI plan able to solve the task via first-principles planning, by resorting to an automated planner (see Figure 9 in Appendix B). This makes failure recovery, via this option, available sometimes.
4. The constraints imposed by deteriorated motors provide situations in which there is less than full coverage.

For a more detailed description of the program, please see Appendix B.

Within the intention structures produced by our program, there are a number of points where either failure recovery or enablement checking may occur. It is at these points that the different intention selection algorithms will differ in their behaviour. Unlike the synthetic tests described in Section 4, there are now opportunities for the agent to recover from failure. If the standard recursive solution fails, the first-principles planning solution can be executed, and if the planned solution becomes blocked, another can be computed. Because $\mathsf{FIFO}^{\mathsf{E}}$ and RR^{E} change focus in order to avoid failure, we expect that they would make less use of failure recovery.

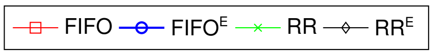
5.2 Experimentation and Results

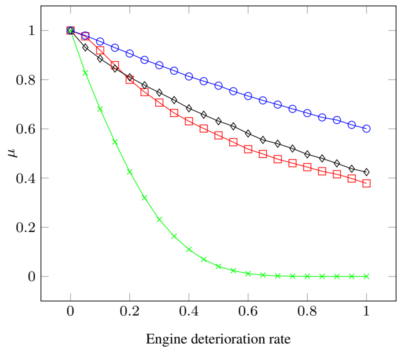
We ran agents with ten 5-disc Towers of Hanoi to solve (i.e., agents with ten intentions). Each test scenario had a randomly selected engine deterioration rate $d \in [0, 1]$ and a randomly selected engine repair rate $r \in [0, 1]$, with 0 representing no deterioration or repair respectively. Each scenario had the same start state of all 5 discs on pin A.

A total of 22,000 scenarios were tested under each of the FIFO, $\mathsf{FIFO}^{\mathsf{E}}$, RR and RR^{E} intention scheduling algorithms. We present the results with respect to the number of successful intentions (effectiveness), the number of moves taken on each tower (efficiency), the amount of failure recovery performed, and the fairness level achieved. For all results which compare the performance of selection algorithms, the significance threshold was set at 0.001.

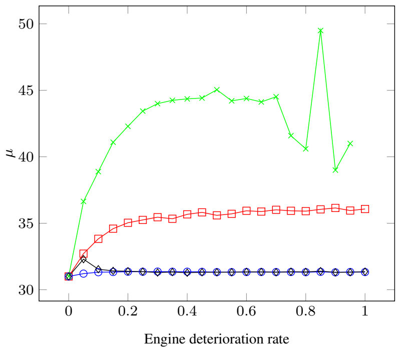
	Success rate		No. moves		Fairness		Recovery rate	
Scheduler	μ	σ	μ	σ	μ	σ	μ	σ
FIFO	0.621	0.434	35.05	7.45	0.424	0.032	0.015	0.017
RR	0.212	0.367	39.92	11.67	0.999	0.001	0.045	0.023
FIFO^E	**0.784**	0.363	31.31	1.21	0.518	0.161	0.005	0.010
RR^E	**0.657**	0.469	31.39	3.04	0.985	0.026	0.015	0.022

(a) Mean (μ) and standard deviation (σ) of all metrics for all algorithms, averaged over all tests.

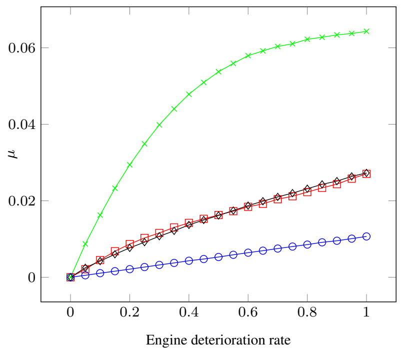




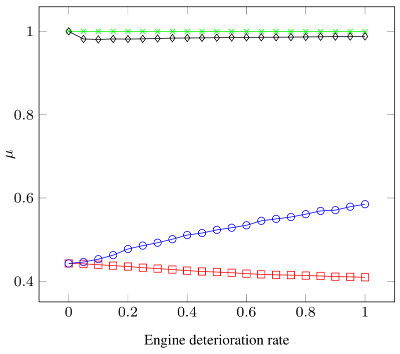
(b) Success rate.



(c) Number of moves per tower.



(d) Failure recovery rate.



(e) Fairness.

Fig. 8 Analysis of all four metrics for all algorithms. Table 8(a) shows mean and standard deviation across all tests. Figures 8(b) – 8(d) depict the mean under different engine deterioration rates.

Success Table 8(a) shows the mean and standard deviation of each algorithm's success rate, averaged over all rates of engine deterioration and repair, while Figure 8(b) shows, for all algorithms, how the mean success rate decreases as the rate of engine deterioration increases (averaging over repair rates). These results are entirely consistent with the synthetic tests, with the two enablement checking algorithms performing better than the standard ones, and with FIFO^E performing the best and standard RR the worst. Differences increase as the adversity of the environment increases.

Efficiency Figure 8(c) shows the difference in number of moves taken by the different approaches. This can be seen as a measure of efficiency, as in general, extra moves, even if they eventually lead to success, will imply extra cost. The number of moves is calculated only for towers successfully solved, as there is no advantage in failing with a small number of moves. The minimum number of moves required to solve a Tower of Hanoi with n discs is $2^n - 1$, and as these tests used 5-disc towers, the minimum moves required is 31. However, in our "extended" version, random motor deteriorations can restrict the moves available to the agent, meaning that it may not be possible to make this optimal sequence of moves. In such cases the agent may need to perform first-principles planning to find an alternative, longer sequence of moves with which to solve the tower. Interestingly, the average number of moves used by the enablement checking FIFO^E and RR^E is always less than 32, even in extremely unpredictable circumstances. By comparison, our tests show that the number of moves taken by standard FIFO and RR increases as the environment becomes less predictable, peaking at 36 and 46 moves respectively.

Failure Recovery While failure recovery is a very important mechanism in BDI agents that ensures their robustness in dynamic worlds, it is nevertheless desirable to minimise its use, as there is likely to be some associated cost. Figure 8(d) shows the differences in use of failure recovery over the course of the execution of all 10 intentions.[12] As would be expected, because the enablement checking approaches change focus when an intention becomes unprogressable, they tend to make less use of failure recovery. For example, when $d = 1.0$, intention selection schemes FIFO^E and RR^E have failure recovery rates of 0.01 and 0.02, respectively, while the rates of FIFO and RR double that, at 0.02 and 0.05, respectively. This difference increases as the environment becomes less stable (i.e., the rate of engine deterioration r increases. With the exception of the static environment, where there is no need for failure recovery, FIFO^E exhibits the least use of failure recovery and RR the highest with FIFO and RR^E very close, much of the time with no statistically significant difference.

Fairness Lastly, Figure 8(e) shows how the different scheduling algorithms compare in term of fairly distributing their attention accross all intentions. As expected, the standard mechanisms RR and FIFO are at the two extremes, with RR maximally fair and FIFO, which maintains focus on an intention until completed, the least fair. Incorporating enablement checking to FIFO improves its fairness to an increasing extent as the environment becomes more adverse, although still well below that of RR or RR^E. RR^E is less fair than RR except in static environments, but only by a very small amount.

As expected, there is a trade-off among desirable properties, in particular between success and fairness. However, as the environment becomes more dynamic, adding enablement checking capabilities to FIFO actually increases its fairness, due to more

[12] As described in Section 3, this is measured as the number of failure recoveries divided by the number of scheduler steps.

focus changes. This means that in dynamic situations, FIFO^E is both fairer *and* more successful than FIFO. Also, incorporating enablement checking into RR substantially improves the success rate of intentions with almost no change in its fairness.

Key Results The key results for this section can be summarized as follows:

1. Consistent with the synthetic test results, incorporating enablement checking in to standard FIFO^E and RR^E always increases success rate, which becomes more pronounced in more adverse environments.
2. FIFO^E and RR^E yield more efficient solutions than their standard counterparts. There is no significant difference between FIFO^E and RR^E, both being highly efficient.
3. Failure recovery is employed least by FIFO^E and most by RR, with FIFO and RR^E showing similar levels.
4. The Round-Robin schemes yield much fairer agents than the First-In-First-Out approaches. Enablement checking makes FIFO^E significantly fairer than FIFO in all but entirely static environments, whereas adding enablement checking to RR decreases its fairness only by a very small amount which remains constant as the environment becomes more adverse.

In summary, incorporating enablenent checking into standard intention selection schemes is completely justified: it is a straightforward addition that yields substantially better results in every metric. In particular, FIFO^E would be the selection mechanism of choice to ensure the maximum number of successful intentions. However, when both success and fairness are desired features, RR^E (that is, Round-Robin with enablement checking) is clearly the preferred approach. Indeed, the drop in success rate from FIFO^E to RR^E is small compared with the large gains in terms of fairness achieved when using RR^E, which yields close to maximally fair behavior. In addition, as we shall discuss below, one would expect to further mitigate the negative interactions between intentions by pairing round-robin approaches with additional strategies for intelligent intention selection.

6 Related Work

The intention selection and management task is crucial to the intelligent behaviour displayed by BDI agents. As a matter of fact, the intention management task—deliberating over what to pursue and how, avoiding conflicting intentions, and striking a balance between commitment to plans and responsiveness to environmental change—has been extensively discussed, from mostly a conceptual perspective, in seminal works in agent theory. In particular, the IRMA architecture [6, 23] for rational agents provided the key reasoning tasks that an intelligent agent should engage in, including what (goals) to pursue (*deliberation* and *filtering* modules) and how (*means-end reasoner* and *opportunity analyser* modules). It turns out that the problem of intention selection in BDI-like systems is the question of how to realize, computationally, many of these reasoning tasks. Indeed, when selecting an intention, a BDI agent may consider which intentions' subgoals to adopt and which plans are applicable for those subgoals. In addition to this, it may seek to avoid potential conflicts among goals and plans, and attempt to exploit positive interaction between them. Thus the development of more powerful intention selection mechanisms, as has been done in this work, directly contributes to the creation of more faithfully rational architectures. The two intention selection mechanisms developed in this paper—enablement checking and low coverage prioritisation—aim at a better integration of means-end reasoning and opportunity analysis into the deliberation task. More concretely, they use the former (how can the agent achieve each goal) to inform the latter (which subgoal to pursue).

Due to a lack of explicit representations of the required mental states, most implemented BDI interpreters implement neither the IRMA compatibility filter nor the filter override (which is responsible for the re-consideration of the agent's current plans). Further to this, the computational complexity required by these reasoning steps makes them impractical for (soft) real-time systems. For example, Grant et al. [15] formalize the semantics of the IRMA's compatibility filter via a set of formal postulates for rationally revising an agent's mental state in the light of new information. Their framework comes with a revision procedure to detect and avoid conflicts between intentions, and commit to (or drop) intentions which have become (in)feasible due to environmental change. However, their technique for checking the rational "optimality" of a set of intentions is provably co-NP hard, and hence not practical for real systems.

An important and related issue much discussed in the agent theory literature is that of *commitment* to goals and plans [9, 24]. The level of commitment adopted by an agent determines the conditions under which it will re-consider its intentions, e.g., a *blindly committed* agent keeps its intentions until they are achieved, while a *single-minded* agent might drop them once it believes that they are unachievable [24]. Rather than abandoning a goal with no applicable plans, enablement checking allows the agent to remain committed, and retry once the means become available. The effect of different commitment levels within a simulated dynamic environment has also been explored by Kinny and Georgeff [21]. Their tests compare the effectiveness of agents with different levels of boldness (i.e., frequency of plan re-consideration) and commitment (i.e., sensitivity to environmental change), the results showing that the most effective agents are bold and rationally committed to their goals. The results presented here also show that in an unpredictable environment, remaining committed while tracking relevant environmental changes can be an effective strategy. However, the deliberation strategies described by Kinny and Georgeff provide the agent with two options—to drop the intention in response to change or to continue regardless. The enablement-checking technique described here provides a third alternative, namely, to remain committed to the selected ends and means, but to temporarily *suspend* execution in the current context.

When it comes to real implemented BDI platforms and programming languages, all but the most simple intention selection decisions have to be directly programmed in agent code. While languages such as Jack and 3APL provide features to aid in this, hand-coding the scheduling and synchronization of intentions can be a complex parallel programming task. The Jack default scheduler provides a choice between FIFO and RR, and while it can be overridden with a custom implementation, it cannot access the agent's mental state. However, Jack provides a number of programming constructs which can be used for this purpose. For instance, the *reasoning statement* `wait_for`(ϕ) blocks an intention until belief ϕ becomes true. In addition, the plan selection and failure recovery processes can be completely overridden or customized via goal-level declarations. By making use of all these advanced features, it is possible, in principle, to encode complex intention management decisions procedurally.

3APL [16] goes further, by allowing the agent's entire deliberation cycle to be directly programmed in a special deliberation-level language. The language includes statements for selecting and applying practical reasoning (PR) rules (akin to plan rules), selecting and executing goals, and expressing preferences over goals and PR rules. A further extension to the language [10] provides constructs to select PR rules based on the agent's belief base, and to perform lookahead planning and re-planning. An appropriate selection of PR rules by a deliberation-level program allows the agent to become more reactive or deliberative, by revising or dropping unachievable goals or blocked plans. Access to the belief base allows the creation of a scheduler which responds to environmental change.

Various agent programming and planning languages have been proposed which allow for scheduling decisions to be made based on programmer-provided information. AgentSpeak(XL) [4] allows the agent structure to be annotated with detailed information such as temporal requirements and deadlines, dependencies and relationships among plans and goals, and the quality, duration, and cost of plans. Through the use of TÆMS *task networks* [12, 17] and the DTC Scheduler [18], the AgentSpeak(XL) interpreter generates an ordering of tasks which satisfies the criteria defined by the various subtask inter-relationships and constraints. The more recent AgentSpeak(RT) [34] has a real-time intention scheduler which allows an agent to schedule time-sensitive intentions in dynamic environments. Plans and actions are extended to include *execution time profiles*, and goals are extended to include (hard or soft) deadlines, priorities, and required confidence levels. The AgentSpeak(RT) scheduler creates an ordering of intentions meeting the subgoals' deadlines, minimum confidence values, and atomicity requirements. When no such ordering is possible, intentions with low priorities may be dropped depending on whether their deadlines are hard or soft. While powerful and expressive, these AgentSpeak extensions rely on non-trivial additional domain information. In contrast, the selection mechanisms described in this paper rely solely on the information already present in typical BDI programs.

Another body of work that is relevant to ours is that of Clement et al. [8] in HTN planners, and the X-JACK language extension described by Thangarajah et al. [30–32] which allows an agent to intelligently handle positive and negative interactions among its plans/intentions. Roughly speaking, both techniques generate summary information at higher levels of abstraction (i.e., higher goals in goal-plan trees) from low level information such as the preconditions and effects of actions and plans' context conditions. Such summary information may indicate that certain conditions must or may hold true during, before, or immediately after the execution of a plan or achievement of a goal. From these summaries it can be determined whether all, some, or none of a goal or plan's possible decompositions will be consistent, or similarly, whether there are any potential conflicts when decomposing two goals for concurrent execution. As a result, an enhanced intention scheduler can use this information to avoid conflicts, prevent the violation of dependency links, and even exploit any positive interaction between plans in a proactive manner. While X-JACK and Clement et al.'s HTN framework do require extra information from the programmer (e.g., preconditions and effects of actions), such information is arguably within the confines of what one could expect within a basic BDI plan library. The positive experimental results shown by their approaches are, in fact, significant in the context of this paper, as they demonstrate the feasibility of an agent making intelligent intention selection decisions based on little more than the bare-bones information required for the functioning of a BDI program. While the intention selection approaches presented in this paper do not explicitly deal with intention interactions issues, they are *orthogonal* to the techniques of Clement et al. and Thangarajah et al.. As such, it possible to integrate their techniques for reasoning about interactions into our schedulers without much difficulty.

7 Discussion and Future Work

In this paper, we have proposed and empirically analysed two techniques for domain-independent intention selection in BDI agent architectures, a key though often neglected aspect of the BDI agent-oriented programming paradigm. Our experimentation clearly demonstrates that *the use of enablement checking when doing intention selection gives a substantial improvement* in the number of successfully completed intentions, and that the use of prioritisation based on know-how coverage improves this further, especially when running vulnerable or highly interrelated programs in volatile environments. In particular, the substantial value of enablement checking is an important result, as it is straightforward to incorporate this into any BDI language/system in the AgentSpeak [25] tradition. Indeed, the operations required to implement it, such as computing the applicable set and evaluating plans' context conditions, are standard elements of this family of agent languages. For example, enablement checking can be forced in Jason [5] by just setting the "events" configuration option to "requeue," which requeues events for which there are no applicable plans. As we have seen from our experiments, incorporating enablement checking to standard intention selection schemes (e.g., FIFO and RR) yields better agent performance in terms of intention success rate and fairness, as well as improvements in terms of efficiency and reliance on failure recovery.

There are a number of promising areas for further work in the areas of enablement checking and low coverage prioritisation, by further refining them or combining them with orthogonal domain-dependent intention management mechanisms that rely on extra information, such as deadlines, priorities, or intention interaction relationships.

Enablement checking The key to the success of the enablement checking approach is that the agent remains strongly committed to the pursuit of its selected goals, even if the means to achieve them are not currently applicable. However, in practical applications, postponing the execution of an intention until a plan is available may have both advantages and disadvantages. Being fast to react, fail the subgoal, and look for alternative ways to accomplish the higher level intention can result in wasted effort or wasted resources, so postponing progression can be advantageous. However, if the intention is urgent, or the environment is unlikely to change, then waiting could be detrimental to the success of the intention. In these cases, a rational agent would not wait for the situation to change for the better, but instead would actively look for an alternative solution via the standard failure recovery mechanisms. Ideally, the agent would adopt an attitude of "*rational commitment*," that is, it would avoid wasted effort by remaining strongly committed to its goals, only reconsidering (or failing) them in appropriate contexts. It is therefore worth investigating a better integration of enablement checking with failure recovery to achieve such a type of commitment. One possibility would be to follow the "flexible" single-minded commitment strategy described in [28], where the agent remains committed to its current (blocked) goal, but only until some alternative plan at a higher level in the goal-plan tree is known to be applicable. In the case of low-coverage prioritisation, the coverage of the alternative plan could further inform the decision between remaining committed and failing the goal. Also, simple domain-dependent priority information can be used by the agent to decide whether a (blocked) intention needs to be recovered or not.

The domains presented in this paper assumed zero cost (and duration) for executing plans and actions. A more sophisticated enablement-checking scheduler would not make this strong assumption. Domain information such as the expected cost of an alternative plan and its subgoals, the cost already spent pursuing active goals, or the expected cost remaining to achieve a non-enabled goal could also be used to inform the agent's decision. Also, enablement checking can be trivially integrated with selection mechanisms based on summary information (e.g., those described in Section 6). Our tests show that as the rate of change of the environemnt decreases, i.e., becomes less likely to change without the agent's intervention, the enablement check becomes less useful. However, in such situations, summary information could allow it to maintain a rational level of commitment by determining if another concurrently executing plan will bring about the conditions required to enable a blocked intention. For example, it may be undesirable to fail a goal because all of its relevant plans require exclusive access to a locked resource when it is known that another concurrent intention is about to release that lock.

Another extension is the use of environmental and temporal information to decide whether to remain committed to a blocked goal. If operating under deadlines, information such as the rate of change, distribution or coverage of relevant environmental variables can be used to calculate the expected time until the intention becomes enabled.

Low-coverage prioritisation The development of more sophisticated coverage calculations is also a promising area for further research. The coverage framework presented here is unable to deal with plans which post goals conditionally, recursively, in parallel, or within loops. However, these are common structures in real-world programs, the Tower of Hanoi program presented in Section 5 being an example. While accommodating conditional and parallel posting of goals requires minimal modification to the formulae, it is not clear how to deal with recursion and looping in BDI plan bodies, since the depth of the recursion and the number of loop iterations is generally not known at compile time. Monte Carlo sampling techniques could possibly be employed to estimate the coverage of such goals and plans. In particular, we believe that it is worth investigating how to extend existing approaches for learning BDI plans' context conditions (e.g., [29]) to learn (and adapt to) the coverage of goals and plans.

Conclusion The addition of generic reasoning mechanisms to improve the "intelligence" of intention selection is a crucial area of research for intelligent agent technology. It is also important, from a practical and scientific point of view, that such approaches do not require substantial additional information to be provided by the developer, and do not compromise the soft real-time requirements of BDI systems through an increase in runtime cost. Both the enablement checking approach as well as the coverage-based intention selection scheme that we have explored in this work meet these criteria, and improve the performance of BDI systems in substantial, measurable ways.

Acknowledgements We acknowledge the support of the Australian Research Council under Discovery Project DP1094627 and Agent Oriented Software for providing us with a Jack license. We would also like to thank the anonymous reviewers for their useful comments. Part of this work was done while the third author was on sabbatical at Sapienza Universita' di Roma, Rome, Italy.

References

1. Steve S. Benfield, Jim Hendrickson, and Daniel Galanti. Making a strong business case for multiagent technology. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 10–15, 2006.
2. R. H. Bordini, L. Braubach, M. Dastani, A. Fallah-Seghrouchni, J. J. Gómez Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.
3. Rafael H. Bordini and Alvaro F. Moreira. Proving BDI properties of agent-oriented programming languages. *Annals of Mathematics and Artificial Intelligence*, 42(1–3):197–226, 2004.
4. Rafael H. Bordini, Ana L. C. Bazzan, Rafael de Oliveira Jannone, Daniel M. Basso, Rosa Maria Vicari, and Victor R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1294–1302, 2002.
5. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-agent Systems in AgentSpeak Using Jason*. Wiley Series in Agent Technology. John Wiley & Sons, 2007. ISBN 0470029005.
6. M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355, 1988.
7. P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents: Components for intelligent agents in Java. *AgentLink Newsletter*, 2:2–5, January 1999.
8. Bradley J. Clement, Edmund H. Durfee, and Anthony C. Barrett. Abstract reasoning for planning and coordination. *Journal of Artificial Intelligence Research*, 28:453–515, 2007.

9. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
10. Mehdi Dastani, Frank S. de Boer, Frank Dignum, and John-Jules Meyer. Programming agent deliberation: An approach illustrated using the 3APL language. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 97–104, 2003.
11. Frank S. de Boer, Koen V. Hindriks, Wiebe van der Hoek, and John-Jules Meyer. A verification framework for agent programming with declarative goals. *Journal of Applied Logic*, 5(2):277–302, 2007.
12. K. Decker and V. R. Lesser. Quantitative Modeling of Complex Environments. *International Journal of Intelligent Systems in Accounting, Finance and Management. Special Issue on Mathematical and Computational Models and Characteristics of Agent Behaviour.*, 2:215–234, January 1993.
13. Michael P. Georgeff and Francois Felix Ingrand. Decision making in an embedded reasoning system. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 972–978, 1989.
14. Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009. ISBN 978-1-58603-929-5. doi: 10.3233/978-1-58603-929-5-633.
15. J. Grant, S. Kraus, D. Perlis, and M. Wooldridge. Postulates for revising BDI structures. *Synthese*, 175:127–150, 2010.
16. Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2 (4):357–401, 1999.
17. Bryan Horling, Victor Lesser, Regis Vincent, Tom Wagner, Anita Raja, Shelley Zhang, Keith Decker, and Alan Garvey. The TAEMS White Paper, 1999. URL `http://mas.cs.umass.edu/paper/182`.
18. Bryan Horling, Victor Lesser, Regis Vincent, and Thomas Wagner. The Soft Real-Time Agent Control Architecture. *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, 12(1):35–92, 2006.
19. Marcus J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proc. of the Annual Conference on Autonomous Agents (AGENTS)*, pages 236–243, 1999.
20. Raj Jain, Dah-Ming Chiu, and William R Hawe. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation, 1984.
21. D. Kinny and M. P. Georgeff. Commitment and effectiveness of situated agents. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 82–88, 1991.
22. Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. Wiley Series in Agent Technology. John Wiley and Sons, 2004. ISBN 0470861207.
23. M. E. Pollack. The uses of plans. *Artificial Intelligence*, 57(1):43–68, 1992.
24. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proc. of Principles of Knowledge Representation and Reasoning*

(KR), pages 473–484, 1991.
25. Anand S. Rao. Agentspeak(L): BDI agents speak out in a logical computable language. In *Proc. of the European Workshop on Modelling Autonomous Agents in a Multi Agent World (MAAMAW)*, pages 42–55, 1996.
26. Anand S. Rao and Michael P. Georgeff. An abstract architecture for rational agents. In *Proc. of Principles of Knowledge Representation and Reasoning (KR)*, pages 438–449, 1992.
27. Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2004.
28. S. Sardina and L. Padgham. A BDI agent programming language with failure recovery, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011.
29. Dhirendra Singh, Sebastian Sardina, and Lin Padgham. Extending BDI plan selection to incorporate learning from experience. *Journal of Robotics and Autonomous Systems*, 58:1067–1075, 2010.
30. John Thangarajah, Michael Winikoff, Lin Padgham, and Klaus Fischer. Avoiding resource conflicts in intelligent agents. In F. van Harmelen, editor, *Proc. of the European Conference in Artificial Intelligence (ECAI)*, pages 18–22, 2002.
31. John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and exploiting positive goal interaction in intelligent agents. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 401–408, July 2003.
32. John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 721–726, August 2003.
33. John Thangarajah, Sebastian Sardina, and Lin Padgham. Measuring plan coverage and overlap for agent reasoning. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1049–1056, 2012.
34. Konstantin Vikhorev, Natasha Alechina, and Brian Logan. Agent programming with priorities and deadlines. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 397–404, 2011.
35. Max Waters, Lin Padgham, and Sebastian Sardina. Evaluating coverage based intention selection. In *Proc. of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 957–964, 2014.
36. Wei Wei and Bart Selman. A new approach to model counting. In *Proc. of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*, pages 324–339. Springer Berlin Heidelberg, 2005.

A Synthetic Domain Details

We provide here further details on how the synthetic testbed used in Section 4 was produced.

Goal-plan trees The goal-plan trees induced by the agent's plan library can be considered perfect binary trees which have had selected branches "pruned" in order to create coverage gaps. We define the depth of a goal-plan tree according to the maximum depth to which subgoals are posted. The top-level goal is at depth 0, a plan relevant to a goal at depth d is also at depth d, and subgoals posted by plans at depth d are at depth $d + 1$.

Gap levels If a goal-plan tree has a *gap level* at depth d, then for every subgoal $!G$ at depth d there is at least one possible world state where $!G$ has no applicable plan. These coverage gaps are modelled on p-effects, i.e., $!G$ is handled by a single plan, $!G : \phi \leftarrow \delta$, and its context condition ϕ is set to *true* in the plan which posted $!G$. The top-level goal has no coverage gaps, so in a tree with depth d any gap layers must be exist between depths 1 and d. So, if a goal-plan tree has a depth of d and g gap levels, the number of possible combinations of gap layers is $\binom{d}{g} = \frac{d!}{(d-g)!g!}$.

In these experiments an agent is tasked with achieving 10 top-level goals, each of which can be decomposed into a binary goal-plan tree with a maximum depth of 4, and 2 gap levels. There are therefore 6 possible combinations of gap levels available. In order to have an even distribution of such structures, the gap layers in each of the 10 goal-plan trees is randomly selected before each test.

Setting the coverage The procedure for setting the coverage of a binary goal-plan tree is shown in Algorithm 1. As can be seen, it is a random, recursive process which sets the coverage of the top-level goal by setting the distributions of the context conditions of the various plans which occur in the tree. The random element ensures that even if two top-level goals have the same coverage, and the goal-plan trees beneath them have the same depth and the same gap layers, the coverage gaps within the trees will still be different.

Algorithm 1 Recursive algorithm for setting a goal-plan tree's coverage

```
procedure SetCoverageOfGoal($G, c$)                          ▷ Sets the coverage of goal $G$ to $c$
    $plans[] \leftarrow$ GetRelevantPlans($G$)
    if $plans.size = 2$ then                                 ▷ This is not a gap layer
        SetDistribution($plans[0].\phi$, 0.5)
        if $plans[0].subgoal \neq null$ then
            $min \leftarrow max(0, c - (1 - c))$
            $c_{!G0} \leftarrow random(min, c)$
            $c_{!G1} \leftarrow 2c - c_{!G0}$
            SetCoverageOfGoal($plans[0].subgoal$, $c_{!G0}$)
            SetCoverageOfGoal($plans[1].subgoal$, $c_{!G1}$)
        else if $c < 1.0$ then
            Error
        end if
    else                                                     ▷ This is a gap layer
        if $plans[0].subgoal = null$ then
            SetDistribution($plans[0].\phi$, $c$)
        else if $plans[0].depth =$ GetDeepestGap then        ▷ No further gaps beneath this goal
            SetDistribution($plans[0].\phi$, $c$)
            SetCoverageOfGoal($plans[0].subgoal$, 1)
        else
            $\alpha_\phi \leftarrow random(c, 1.0)$
            $c_{!G} \leftarrow c/\alpha_\phi$
            SetDistribution($plans[0].\phi$, $\alpha_\phi$)
            SetCoverageOfGoal($plans[0].subgoal$, $c_{!G}$)
        end if
    end if
end procedure
```

Test steps For each test, the following steps are performed:

1. 10 different top-level goals are randomly selected from the agent's library.
2. A random coverage level $c \in [0.01, 0.99)$ is selected.
3. The coverage of each of the 10 top-level goals is set to c.

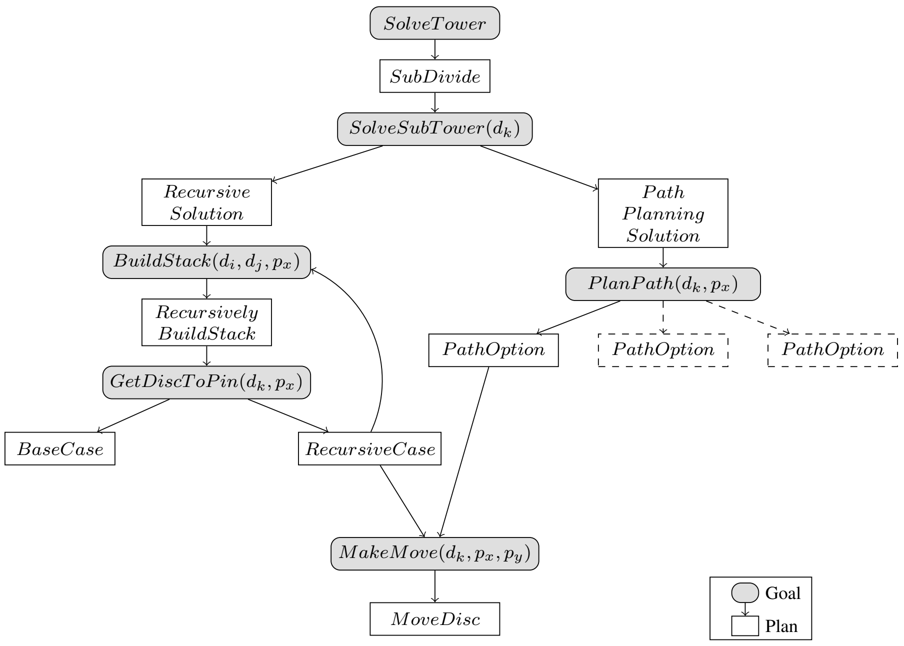


Fig. 9 The goal-plan tree structure for the Tower of Hanoi agent.

4. A random environmental dynamism level $d \in [0.0, 1.0)$ is selected.
5. The initial state of the environment is set by resampling each environment variable, as per each probability distribution. This state is then saved.
6. For each intention selection mechanism being tested:
   (a) The 10 top-level goals selected in step 1 are posted, and the results of the agent runs are recorded.
   (b) The environment is re-set to the state saved in step 5.

B Tower of Hanoi Agent Design

The standard Tower of Hanoi puzzle consists of three pins, p_A, p_B and p_C, and n discs, $d_1, d_2, \ldots, d_n$, numbered according to their size, with d_1 being the smallest and d_n the largest. In the game's initial state, all of the discs are placed on to pin p_A in order of size, i.e., with disc d_n at the bottom of the stack and d_1 at the top. The goal-plan tree structure used by the agent to solve a single instance of the Tower of Hanoi is shown in Figure 9. In our tests, the agent is tasked with solving several Tower of Hanoi instances simultaneously. Therefore, at any point in time, the agent's intention base may comprise several partially-executed instances of this intention.

The goal-plan tree's top-level goal, $SolveTower$, is handled by a single plan, $SubDivide$, which divides the Tower of Hanoi problem into a series of smaller sub-problems. The branch stemming from the plan $RecursiveSolution$ is a BDI implementation of a recursive solution to the Tower of Hanoi, while the alternative branch, $PathPlanningSolution$, solves the tower by searching the state-space and planning a sequence of moves. We will now describe each component of the goal-plan tree in more detail (note that when referring to discs, d_n will always refer to the largest disc, while all other subscripts denote variables).

Moving discs The goal $MakeMove(d_k, p_x, p_y)$ is posted by the agent whenever it needs to move disc d_k from pin p_x to pin p_y, and is handled by a single plan, the primitive action $MoveDisc$. The context condition of $MoveDisc$ first checks that the move is legal, i.e., that disc d_k is at the top of p_x, and is smaller than the disc at the top of p_y. It then checks that the move is available, given the constraints

Algorithm 2 Recursive solution to the Tower of Hanoi

```
procedure BuildStack($d_i, d_j, p_x$)                    ▷ Stacks discs $d_i, d_{i-1} \dots d_j$ onto $p_x$
    for $k \leftarrow i, k \geq j, k \leftarrow (k-1)$ do
        GetDiscToPin($d_k, p_x$)
    end for
end procedure

procedure GetDiscToPin($d_k, p_x$)
    $p_y \leftarrow$ GetCurrentPin($d_k$)
    if $k = 0 \vee p_x = p_y$ then                                ▷ Base case
        return
    else                                                   ▷ Recursive case
        $p_{tmp} \leftarrow$ GetOtherPin($p_x, p_y$)
        RecursivelyBuildStack($d_{k-1}, 1, p_{tmp}$)
        MoveDisc($d_k, p_y, p_x$)
    end if
end procedure
```

imposed by the current state of the tower engines. If these conditions hold, then the plan will be selected and the disc moved, if not, then the $MakeMove$ goal will have no applicable plans.

Solving sub-problems The traditional solution to the Tower of Hanoi is recursive, i.e., to solve a tower of n discs, it first solves a tower of $n-1$ discs. The base case, a tower of just one disc, can be solved trivially. The plan $SubDivide$ sub-divides a Tower of Hanoi into sub-problems in a similar way. The aim of this plan is to stack discs $d_n, d_{n-1}, \dots, d_1$ (where d_n is the largest and d_1 the smallest) on to pin C. To achieve this it first attempts to stack d_n on to p_C, and then both d_n and d_{n-1}, etc. The subgoal $SolveSubTower(d_k)$ represents the need to create a stack of discs, $d_n, d_{n-1}, \dots, d_k$, on to p_C. $SubDivide$ therefore synchronously posts n instances of this goal, i.e., $SolveSubTower(d_n)$, $SolveSubTower(d_{n-1}), \dots, SolveSubTower(d_1)$.

Recursive solution The agent's preferred techique for solving a stacking sub-problem is a modification of the traditional recursive solution. From an arbitrary initial state, this solution will build a stack of any given size on any pin using the fewest possible moves. The pseudocode for this algorithm is shown in Algorithm 2. For any values of i, j such that $1 \leq j \leq i \leq n$, the procedure will stack discs $d_i, d_{i-1}, \dots, d_j$ onto a given pin p_x.

The structure of this algorithm is reflected directly in the structure of the goal-plan tree in Figure 9. Calls to procedure BuildStack are represented by the goal of the same name, and the procedure body is implemented in the plan $RecursivelyBuildStack$. Calls to procedure GetDiscToPin are represented by the goal of the same name, with its base and recursive cases implemented in plans $BaseCase$ and $RecursiveCase$. The plan $RecursiveSolution$ handles a subgoal of the form $SolveSubTower(d_k)$, which, as described above, represents the need to stack discs $d_n, d_{n-1}, \dots, d_k$ onto p_C. The plan thus posts a single subgoal, $BuildStack(d_n, d_k, p_C)$, which effectively "calls" procedure BuildStack, beginning the recursive process.

The goal $BuildStack(d_i, d_j, p_x)$ represents the need to build a stack of discs, $d_i, d_{i-1}, \dots, d_j$, on pin p_x. The plan $RecursivelyBuildStack$ handles this goal, and achieves it by first getting disc d_i on to the top of pin p_x, then placing disc d_{i-1} on top of d_i, etc., until the stack is complete. It does this by posting a series of goals of type $GetDiscToPin$.

As the name implies, the goal $GetDiscToPin(d_k, p_x)$ represents the desire to place disc d_k on to the top of pin p_x. It is handled by two plans, $BaseCase$ and $RecursiveCase$. The context condition of $BaseCase$ is simply the boolean condition of the if-then construct in procedure GetDiscToPin, i.e., it checks to see if the goal $GetDiscToPin$ has already been achieved. Thus, if selected, this plan need not do anything. The context condition of $RecursiveCase$ is the negation of the same boolean condition. This plan first finds the "temporary" pin, p_{tmp}, i.e., the pin which is neither the destination pin, p_x, nor the pin which is disc d_k's current location. It then posts a subgoal, $BuildStack(d_{k-1}, d_1, p_{tmp})$. The purpose of this recursive posting is to stack all discs smaller than d_k onto the "temporary" pin, thus clearing the way for d_k to be moved on onto p_x. This move is achieved by posting a $MakeMove$ subgoal.

While simple and efficient, this method is brittle. When following this shortest route, it pays no attention to the current state of the engines, meaning that it may post a $MakeMove$ goal that is not currently achievable. This might (depending on the intention selection scheme used) cause a goal failure which will propagate up to the $RecursiveSolution$ plan.

Path planning solution If the recursive solution fails, the agent has an alternative, path-planning based way of resolving the $SolveSubTower$ goal. The plan $PathPlanningSolution$ handles a goal of the form $SolveSubTower(d_k)$, and simply posts a single goal, $PlanPath(d_k, p_C)$.

The goal $PlanPath(d_k, p_x)$ represents the need to plan a path to a state in which $d_n, d_{n-1}, \ldots, d_k$ are stacked on p_x. When it is posted, a breadth-first search over the space of all possible moves is performed and, given the current state of the engines, the shortest sequence of moves to all tower states which satisfy the goal are found. A separate instance of the plan $PathOption$ is generated per sequence, and the plan instance with the shortest sequence is then selected for execution. The $PathOption$ plan follows the path by posting a series of $MoveDisc$ goals.

It is of course possible that while the path is being followed, the engine deteriorates in such a way as to block the path. In this case, the $PathOption$ plan might fail while attempting an unavailable move. But rather than this failing the intention as a whole, this failure will prompt the goal $PlanPath$ to be re-posted, and an alternative path will be sought. However, it is still posible for the path-planning solution to fail. If the engines are sufficiently deteriorated, it is possible that no desired states are accessible; in this case no applicable plans will be generated for the goal $PlanPath$. Depending on the intention selection scheme in place, this may cause a goal failure which will propagate to the intention's top-level goal.