BDI Agent Architectures: A Survey

Lavindra de Silva[1] , **Felipe Meneguzzi**[2] and **Brian Logan**[3]
[1] University of Cambridge, Cambridge, UK
[2] Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil
[3] University of Nottingham, Nottingham, UK
lavindra.desilva@eng.cam.ac.uk, felipe.meneguzzi@pucrs.br, brian.logan@nottingham.ac.uk

Abstract

The BDI model forms the basis of much of the research on symbolic models of agency and agent-oriented software engineering. While many variants of the basic BDI model have been proposed in the literature, there has been no systematic review of research on BDI agent architectures in over 10 years. In this paper, we survey the main approaches to each component of the BDI architecture, how these have been realised in agent programming languages, and discuss the trade-offs inherent in each approach.

1 Introduction

For the last 30 years, the BDI agent model based on the mental attitudes of beliefs, desires and intentions has formed the basis for much of the research on architectures for autonomous agents. Starting with the philosophical work of Bratman [1987], and implementations such as the Procedural Reasoning System (PRS) [Georgeff and Lansky, 1987], the theory and practice of BDI agents has proceeded in parallel, with innovations at the semantic level leading to new architectural and language features, and new architectural features leading to new and extended semantics. Over this period, many agent architectures, languages, interpreters, platforms, and theoretical formalisations have been developed, embodying a wide range of agent programming features and their corresponding semantics. The resulting space of theories and implementations is complex, and a review of the current state of the BDI ecosystem seems timely.[1]

In this paper we survey the most important features of BDI agent architectures and their implementations in agent programming languages. While the importance of a feature is inevitably subjective, our survey is guided by two key criteria. First, a feature must fall squarely within the original BDI concept described by Bratman [1987]. This criterion excludes architectures that contain additional mental attitudes such as obligations. Second, it must either implement or facilitate the implementation of Bratman's notion of practical reasoning. This excludes features targeting, e.g., software development issues, such as modules and debugging support. Given the space available and the volume of work in the literature, we are only able to give representative examples for many features we consider, and we apologise to the authors whose work we had to omit.

[1]The most recent survey of which we are aware is [Bordini *et al.*, 2006]; [Kravari and Bassiliades, 2015] focuses on the pragmatics of agent platforms, e.g., license and robustness rather than architectures per se.

2 The BDI Agent Architecture

The BDI agent architecture has its origins in the philosophical work of Bratman [1987]. Bratman argued that practical reasoning can be thought of as the act of weighing multiple, conflicting considerations for and against conflicting choices, in light of what the agent believes, desires, values and cares about. The first step in practical reasoning, known as *deliberation*, is to decide *what* state of affairs to bring about from the (possibly conflicting) desires of the agent, and the second step, known as *means-ends reasoning*, is to decide *how* to bring about that state of affairs. In practice, means-ends reasoning concerns the adoption of some 'recipe' or pre-specified plan of action in order to bring about an intended state of affairs. Bratman's philosophical model inspired the idea of programming intelligent agents based on the mental attitudes of beliefs, desires and intentions.

In a BDI architecture, beliefs represent the agent's information about its environment, other agents and itself, goals (desires) are states of affairs to achieve, and intentions are commitments to achieving particular goals. A BDI agent program consists of a set of initial beliefs (and in some cases goals) and a set of plan-rules (analogous to methods in HTN planning) specifying when a plan can be used to achieve a goal or respond to changes in the agent's beliefs. Plans consist of a sequence of steps, and may include belief updates, actions and subgoals. The execution of an agent follows a deliberation cycle implemented by an agent interpreter (Algorithm 1) that realises both deliberation and means-ends reasoning. The cycle begins with the agent updating its event queue with any external events (e.g., new top-level goals or percepts) from the environment and any internal events (e.g., subgoals) from the previous cycle (Line 3). Any percepts (i.e., changes in the state of the environment) are then used to update the agent's beliefs, giving an updated belief base Bel (Line 4). The agent then uses the plan-rules comprising its plan library to respond to new events. Each plan-rule proposes a plan to respond to an event based on the agent's current beliefs. If the event is

Algorithm 1 BDI Agent Interpreter

```
1: procedure AGENTINTERPRETER($\langle Ev, Bel, PLib, Int\rangle$)
2:     while $true$ do
3:         $Ev \leftarrow$ UPDATEEVENTS($Ev$)
4:         $Bel \leftarrow$ UPDATEBELIEFS($Ev, Bel$)
5:         $Int \leftarrow$ SELECTPLANS($Ev, Bel, PLib, Int$)
6:         $Ev \leftarrow$ EXECUTEINTENTION($Int, Ev$)
```

external, the plan forms the root of a new intention; if the event is a subgoal, the plan is added to an existing intention (Line 5). Each intention thus comprises a stack of plans adopted to achieve a hierarchy of subgoals. Finally, the agent executes the next step of the topmost plan in an intention. This may involve updating the event queue with a new subgoal, or executing an action in the environment (Line 6). The cycle then repeats.

Algorithm 1 abstracts away from the detail of how each step in the cycle is implemented, but illustrates the key features of the BDI architectures and languages we consider in this paper. For a more detailed exploration of a computable BDI interpreter, see Meneguzzi and De Silva [2015]. However, while the basic components of the BDI architecture and steps in the deliberation cycle have remained largely constant throughout the history of research into BDI agents, they have been elaborated and interpreted in a variety of ways as we describe in the following sections.

3 Approaches to Beliefs

In most agent programming languages, an agent's belief base is represented as a conjunction of ground positive literals in a first-order logical language. However, in many cases, this basic representation is enriched in various ways as described below.

3.1 Belief Formulas

In many languages, the beliefs of an agent represent only 'true facts' (i.e., positive literals). This has the advantage of ensuring that the agent's beliefs are always consistent; however, representing negative information, e.g., that there is no block on top of $block1$, typically requires additional fluents, e.g., $clear(block1)$. In such languages, negation (if it is supported at all) is only allowed in plan-rules in the form of negation as failure. That is, the agent implicitly believes $not(\phi)$ if ϕ is not present in (or derivable from) the agent's belief base. However, some languages, e.g., Jason [Bordini *et al.*, 2007], allow negative literals in the agent's belief base (termed 'strong negation' in Jason). In addition, many languages allow the agent's beliefs to include inference rules (typically Horn clauses). Such rules are usually viewed as representing fixed, 'definitional' information about a domain, which is not updated by the agent's percepts or messages from other agents.

3.2 Extending the Belief Representation

There has also been work on extending the underlying belief representation to support more complex forms of inference. For example, Moreira *et al.* [2006] develop a variant of AgentSpeak(L) based on description logic rather than Prolog. This allows complex belief base queries to be expressed more concisely, consistency checking of belief updates, more flexible retrieval of plans based on the subsumption relation between concepts, and the sharing of knowledge expressed in ontology languages such as OWL. A similar approach was taken in the JASDL extension of the Jason agent platform that makes use of an existing OWL-API to provide features such as plan trigger generalisation based on ontological knowledge and the use of such knowledge in querying the agent's belief base [Klapiscak and Bordini, 2009].

Another strand of work seeks to avoid a commitment to any particular knowledge representation (KR) technology, and instead aims to make the underlying KR technology a 'pluggable' component. For example, Dastani *et al.* [2009] develop two different approaches to integrating and combining multiple KR techniques into a BDI-based agent programming language: a meaning-preserving translation approach that maps one representation to another, and an approach based on 'bridge rules' which adds additional inference power to an agent system with multiple KR technologies. An alternative approach to the integration of KR languages is to use a generic KR interface, which can be used to integrate, e.g., both Prolog and OWL with rules [Bagosi *et al.*, 2015]. On a practical level, some languages, e.g., Jason, allow additional information to be stored as 'annotations' on beliefs, e.g., the source of the belief and the timestep at which it came to be believed, which may be used by other forms of reasoning or belief update procedures.

3.3 Uncertain Beliefs

There has also been work on extending BDI languages to handle uncertain beliefs. For example, Kwisthout et al. [2005] used Dempster-Shafer theory to model uncertainty in an agent's beliefs; Casali et al. [2011] proposed a graded BDI agent model in which belief degrees represent the agent's certainty about beliefs; Silva and Gluz [2011] presented a variant of AgentSpeak(L) [Rao, 1996], AgentSpeak(PL), that supports probabilistic beliefs through the use of Bayesian Networks; and Bauters et al. [2014] extended the operational semantics for the agent language CAN [Winikoff *et al.*, 2002] to deal with uncertain information. Other approaches separate the symbolic BDI cycle from Bayesian update of percepts, e.g., Coelho and Nogueira [2015], where the update function uses a hidden Markov model internally but returns the most probable percept as a symbolic element of the belief base.

3.4 Belief Revision

There has also been work on how an agent's beliefs are updated. For example, Alechina *et al.* [2006] have investigated how AI belief revision techniques can be used to ensure consistency of a BDI agent's belief base. Another strand of work investigates how argumentation theory can be used to reason about contradictory information, e.g., in multi-agent interactions. For example, Panisson *et al.* [2018] have developed techniques to allow Jason/JaCaMo agents to choose between conflicting conclusions, or to choose the most promising arguments in a dialogue by considering information sources of varying degrees of trustworthiness.

4 Approaches to Goals

Goals are another key concept in BDI agent programming. In this section, we briefly review the most important approaches to goals proposed in the literature, including the main types of goals and their semantics.

4.1 Test Goals

A test goal is a goal to test whether a condition holds (is believed by the agent). In most BDI programming languages, test goals occur only as (sub)goals in plans, and not as top-level goals. A test goal is evaluated against the agent's belief (or goal) base, possibly unifying variables. If the goal evaluates to true, execution of the plan containing the goal continues. However, languages take differing approaches to test goals that evaluate to false. For example in 2APL [Dastani, 2008] execution of the plan containing the goal blocks until the goal evaluates to true (in PRS this is achieved using the ?WAIT construct), while in Jason, the failure of a test goal may trigger a subgoal and the adoption of a plan to make the test true, e.g., by asking other agents for information.

4.2 Achievement Goals

Achievement goals can be categorised into two main types. Procedural achievement goals are 'goals to do', i.e., a commitment to perform an action or sequence of actions in response to a change in the agent's goals or beliefs. Declarative achievement goals, on the other hand, are goals 'to be', meant to bring about a particular state of the environment or agent. Declarative goals are intrinsically related to the agent's beliefs: a rational agent should not have as a goal a state which it currently believes to be the case. If the agent comes to believe that the goal state is currently the case, it will drop the goal. In contrast, procedural goals are generally independent of the agent's beliefs, in the sense that no change in the agent's beliefs will result in the agent dropping the goal.

Different agent programming languages typically support either procedural or declarative goals but not both. For example, languages such as PRS, JACK [Busetta *et al.*, 1999], 3APL [Hindriks *et al.*, 1999] Jason and Jadex [Braubach *et al.*, 2005] take a procedural view of achievement goals, while languages such as GOAL [Hindriks *et al.*, 2001], later versions of 3APL [Dastani *et al.*, 2004] and 2APL take a declarative view. Some languages that take a procedural view support declarative goals indirectly through programming idioms. For example, Hübner *et al.* [2006] show how declarative goals with both success and failure conditions can be implemented in Jason using plan-rule patterns.

In languages that take a procedural approach to goals, goals are typically represented by simple terms, possibly containing variables, e.g., $pickup(block1)$. If the agent has multiple goals, they may be performed in any order or simultaneously, depending on the capabilities of the agent and its program. In languages that support declarative goals, goals are typically represented by conjunctions of positive literals, possibly containing variables. For example, $on(block1, block2) \wedge on(block2, block3)$ specifies that $on(block1, block2)$ and $on(block2, block3)$ should be true in the same state. Each goal is assumed to be consistent, i.e., the agent cannot have $\phi \wedge \neg\phi$ as a goal. However, if the agent has multiple goals, they do not need to be consistent, i.e., achieved simultaneously.

In some languages, e.g., PRS, CAN and Jadex, the specification of an achievement goal may also include a 'failure condition' specifying the situations in which the goal is considered unachievable or irrelevant and should be dropped. Goals may also be dropped when the plan adopted to achieve them fails, though this is usually determined by the deliberation cycle as described in Section 6.

Both procedural and declarative goals are typically considered 'adopted', in the sense that the agent will pursue/achieve them at some point in the future. However, the decision about when a goal is achieved is typically left to the deliberation cycle as discussed in Section 6. As such, they function more like desires in BDI logics.

4.3 Maintenance Goals

A maintenance goal is a goal to maintain a particular state of the environment. As with (declarative) achievement goals, maintenance goals give rise to the adoption of plans when the goal condition does not hold, or when a guard condition (termed a maintain condition in [Dastani *et al.*, 2006]) becomes true, which implies the state to be maintained will cease to hold in the near future. For example, a goal to maintain a charged battery may be activated by the charge level dropping below 20%. In contrast to achievement goals which may be achieved in the future, the guard condition becoming true or the goal condition becoming false typically results in the immediate adoption of a plan to re-establish or maintain the goal condition. However, similar to achievement goals, if the plan fails, how the failure is handled is determined by the deliberation cycle as described in Section 6.

4.4 Temporally Extended Goals

Both (declarative) achievement and maintenance goals can be seen as special cases of temporally extended goals. A temporally extended goal is specified by a formula in temporal logic. In this setting, achievement goals can be seen as reachability properties (in the future ϕ holds), maintenance goals as invariant properties (ϕ holds in all states), and the active preserve goals of PRS as similar to Until formulas (ψ holds until ϕ holds, although in PRS ϕ is a procedural goal). Dastani *et al.* [2011] present a rich taxonomy of goal types, where the goals are represented by LTL formulas which are translated into combinations of more basic declarative and maintenance goals. For example, an 'achieve and maintain' goal can be defined as one which first achieves a certain condition (as in a declarative achievement goal) and then maintains it over a certain period of time. Dastani *et al.* sketch how such temporal goals can be realised in standard agent programming languages; however, with the exception of the special cases noted above and preliminary work in [Hindriks *et al.*, 2009], temporal goals are not currently supported by most agent programming languages.

4.5 Priorities, Deadlines and Plan Restrictions

In addition to the main goal types listed above, various additional properties and constraints on goals have been proposed in the literature. These have mostly been explored in the context of declarative goals, but in principle there is no reason why they cannot be be applied to procedural goals, and, in some cases to maintenance goals. For example, AgentSpeak(XL) allows the utilities and deadlines of goals to be specified, and the agent acts to maximise an objective function [Bordini *et al.*, 2002]. Similarly, AgentSpeak(RT) allows both the priority of a goal and deadline by which it should be achieved to be specified, and, if not all goals can be achieved (by their deadlines), an AgentSpeak(RT) agent will pursue a priority maximal set of goals that can be achieved by their deadlines [Vikhorev *et al.*, 2011].

5 Approaches to Plans

A plan-rule typically comprises a trigger specifying when the rule is relevant, a context condition specifying when the rule is applicable (i.e., which beliefs the agent must have for the plan to be appropriate), and a plan built from steps such as goals, belief update operations, and actions. In the simplest case, the steps comprising a plan are executed in sequence. However, several languages provide constructs for controlling the order in which the steps in a plan are executed which we discuss in this section.

5.1 Series-Parallel Interleaving

In languages that support series-parallel interleaving, plans are built incrementally by the sequential and parallel compositions of other plans. One approach is to interleave execution of the plan for a subgoal with the remaining steps in the plan containing the subgoal by treating the subgoal as a separate intention. For example, in Jason, the plan $!!e; a_1; a_2$ specifies that the subgoal e should be treated as a new intention, allowing the execution of the plan that achieves e to be interleaved with the execution of $a_1; a_2$, i.e., the plan for e may execute before, after or during the execution of $a_1; a_2$, and is completely independent of it. In particular the failure of the plan for e is not considered a failure of the invoking plan.

Some languages, e.g., CAN and AgLOTOS [Chaouche *et al.*, 2014], allow finer control over the ordering for steps. For example, the steps in the plan $(a_1; a_2) \mid (a_3; a_4)$ can be executed in any order (e.g., in the order $a_1 \cdot a_3 \cdot a_2 \cdot a_4$) provided a_2 happens after a_1, and a_4 happens after a_3. In AgLOTOS, the interleaving of steps can also be synchronised. For example, in the program $(a_1; \overline{a}_2; a_3) \mid (a_4; \overline{a}_2; a_5)$ where $\overline{a}_2$ is a 'synchronisation step', a_1 and a_4 can execute in any order, but the two branches must then synchronise via action $\overline{a}_2$ before the remaining actions can be executed.

5.2 Arbitrary Interleaving

Languages that support arbitrary interleaving allow more fine grained ordering of steps than series-parallel interleaving. For example, consider the series-parallel plan $P = !e_1; \big(!e_5 \mid (!e_2; !e_6)\big); !e_3; !e_4$, which is the sequential composition of the subgoal e_1, interleaved plan $(!e_5 \mid (!e_2; !e_6))$, and sequential plan $!e_3; !e_4$ (where $!e$ indicates that the subgoal e needs to be achieved). The interleaved plan specifies the interleaving of the plan to achieve e_5 and sequential plan $!e_2; !e_6$. Observe that it would not be possible to modify P to specify that e_6 (i.e., the plan chosen to handle it) should additionally be interleaved with e_3 (i.e., the plan chosen to handle it) without violating the other ordering requirements (or the need to leave certain steps unordered). However, systems that support arbitrary interleaving (e.g., PRS [de Silva *et al.*, 2018] and HTN-acting [de Silva, 2018]) allow specifying exactly the ordering requirements in P as well as the interleaving of e_6 and e_3. For example, this is done in HTN-acting by representing P as a partially ordered set of labelled subprograms.

5.3 Non-Interleaving (True) Concurrency

Some languages support true concurrency, i.e., where steps may be executed simultaneously. For example, the plan $P \parallel P'$ allows steps in P to be interleaved or executed simultaneously with those in P', e.g., on different processors. In JACK a concurrent plan may be specified to execute in one of four ways [de Silva, 2020]: *(i)* if a branch fails, the plan is considered to have failed — the other branches are notified and may perform 'cleanup' steps before terminating; *(ii)* if a branch fails, the plan is considered to have failed but the remaining branches are allowed to complete execution; *(iii)* if a branch completes execution, the plan succeeds – the other branches are notified and may perform 'cleanup' steps before terminating; and *(iv)* if a branch completes execution, the plan succeeds but the remaining branches are allowed to complete execution. Other languages support fewer forms of concurrency. For example, the Concurrent CAN operational semantics supports the second approach [de Silva, 2020], and OpenPRS supports the first directly [Ingrand, 2014] (and the second through programming idioms). JAM supports a more basic approach to true concurrency called 'lock step concurrency', where the first step in each branch is executed simultaneously, and then the same is done with the second step in each branch, and so on [Huber, 2001].

5.4 Control Flow

In addition to the various forms of parallelism described above, some languages such as Jadex, OpenPRS, the Refinement Acting Engine (RAE) [Ghallab *et al.*, 2016], JACK, 3APL, and Jason support conditional and looping constructs found in imperative programming languages, e.g., `if-then-else` statements and `while` loops. However, such constructs are essentially 'syntactic sugar' and do not increase the expressive power of a BDI language, as the same effect can be achieved using multiple plan-rules with different context conditions. For example, to specify the equivalent of `if` ϕ `then` a_1 `else` a_2 in languages such as AgentSpeak, we can specify two plan-rules of the form $e : \phi \leftarrow a_1$ and $e : \neg\phi \leftarrow a_2$, where achieving subgoal e represents executing the `if` statement. Moreover, a richer plan syntax may make it more difficult to implement some of the extensions to the BDI deliberation cycle discussed in the next section.

6 Approaches to the Deliberation Cycle

Finally, we discuss approaches to the deliberation cycle. We focus on failure handling, intention scheduling for avoiding or exploiting interactions between intentions, and meta-level reasoning for implementing sophisticated agent behaviour, as these have received most attention in the literature.

6.1 Failure Handling

In some BDI systems, failure recovery is manual in the sense that it needs to be programmed by the developer via 'failure' plan-rules or procedures. However other languages support various forms of automated failure recovery, i.e., the failure recovery mechanisms are built into the behaviour of the system.

Manual Failure Recovery. In some languages, e.g., JAM, recovery is performed by 'failure procedures' associated with each plan. A failure procedure is called when the associated plan fails during execution, e.g., because an action in the plan fails or because there are no applicable plan-rules for a subgoal. Failure procedures are mainly intended for specifying 'cleanup steps': they cannot include subgoals and are executed atomically (i.e., execution is not interleaved with execution of other intentions). Other languages, e.g., Jason, treat recovering from a failure as a particular kind of goal to be achieved. The failure of a plan for a goal $!e$ causes the associated intention to be suspended and a corresponding 'goal deletion' event $-!e$ to be posted. Goal deletion events trigger 'failure' plan-rules, where the plan typically contains 'cleanup steps', possibly followed by the re-posting of the goal that failed.

A more sophisticated approach adopted by, e.g., 3APL and Jadex, is to use special failure plan-rules to perform meta-level reasoning about the intention that failed. Such rules are triggered by (partial) plans rather than goals or beliefs. For example, a rule whose head matches a sequence of plan steps beginning with a failed step may replace the failed step and some or all of the subsequent steps, effectively rewriting the plan at run time. Handling failure using meta-level reasoning allows more informed recovery than the approaches above, as there is access to information that is not in the belief base, e.g., the (remaining) steps in an intention.

Automated Failure Recovery. The most basic approach to automated failure recovery is to repeatedly retry the failed action until it succeeds [Wobcke, 2001]. However, in most languages, e.g., CAN, JACK, RAE, and HTN-acting, automated failure recovery happens at the level of goals. That is, if a plan fails to achieve its triggering goal $!e$, an alternative applicable plan-rule is selected from the ones that are relevant for $!e$. If no alternatives exist, or the plans for all alternative applicable rules also fail, the agent then backtracks to $!e$'s 'parent' goal (i.e., the triggering goal of the plan that posted $!e$ as a subgoal), in order to try an alternative applicable rule at that level. This backtracking process may continue until the top-level goal is reached. Three main approaches to selecting an alternative applicable plan-rule have been proposed in the literature: *(i)* in some languages, e.g., JACK, plan-rule context conditions can be checked 'early', i.e., immediately after they are added to the set of relevant rules for a goal, in order to instantiate the applicable rules and store them for use later upon failure; *(ii)* in languages such as e.g., CAN and RAE, plan-rule context conditions are (re)checked only after failure occurs and an applicable alternative is required (which may be well after the set of relevant rules was created); and *(iii)* in HTN-acting, plan-rule context conditions are checked immediately before the first step in the plan is executed.

However, in HTN-acting, a plan-rule is checked (for applicability) only once, i.e., the rule is not re-checked on the failure of another plan for the goal. A more 'complete' account is given as part of the second approach to failure above, which re-checks a plan-rule unless an instance of its plan has failed (as in, e.g., CAN), or re-checks plan-rule *instances* whose plans have not failed (as in, e.g., JACK, RAE, and the CAN variant in [Sardina and Padgham, 2011]). An even more complete account, as stated in [Ghallab *et al.*, 2016], would be to re-check plan-rule instances whose plans *have* failed, if the conditions responsible for the failure no longer hold.

6.2 Intention Scheduling

Intention scheduling is the process of suitably interleaving the intentions of an agent. The most basic approaches involve executing one step in an intention at each deliberation cycle, in either a FIFO or round robin manner as in, e.g., JACK and Jason, or an intention with the highest utility as in JAM, where the utility of an intention is computed from the utilities of the goals the intention achieves and the plans used. However, such simple approaches can result in conflicts between intentions (where the execution of an action in one intention destroys the precondition of an action in another intention) and/or 'wasted effort' (e.g., where the same subgoal is achieved by multiple intentions), and several approaches to intention scheduling have been proposed that focus on exploiting positive interactions between interleaved intentions, and/or avoiding negative interactions between intentions or between intentions and the environment.

Exploiting Positive Interactions. When the next action in an intention cannot be executed because the preconditions of the action do not hold at the current deliberation cycle or there is no applicable plan for a subgoal, it may be possible to progress a different intention that has as a 'side effect' the (re)establishment of the preconditions of the action or context condition in the blocked intention. This is termed a positive interaction, and differs from the failure handling approaches discussed above, where the plan containing the blocked action would be aborted (perhaps after performing some 'cleanup' steps), and an alternative applicable plan-rule selected for the corresponding goal. To exploit such positive interactions, Yao *et al.* [2016] proposed a scheduling approach based on Monte-Carlo Tree Search, in which pseudorandom simulations of different interleavings of the steps within each intention are used to determine which intention to progress next.

Avoiding Negative Interactions. A negative interaction occurs when a step that is executed in one intention brings about an effect that causes a condition in another intention to become violated. To avoid such conflicts, Thangarajah et al. [2011] compute 'summary' information offline from the agent's goals and plan-rules that describes the necessary and possible pre- and post-conditions for different ways of achieving a goal. This information is used at runtime to schedule intentions, by checking whether a newly adopted goal will definitely be safe to execute without conflicts, definitely result in conflicts, or possibly result in conflicts. If the goal cannot be achieved without conflicts the corresponding intention is deferred. Waters et al. [2014] compute an intention's 'coverage', i.e., how complete the available plan-rules are for achieving the intention, relative to the set of possible environmental situations. Their intention scheduler prioritises intentions with low coverage, i.e., those with a high chance of becoming non-executable due to changes in the environment. However, in their approach and in that of Thangarajah et al. [2011], intentions are scheduled at the plan level. Yao et al. [2016] show how an approach based on Single-Player Monte-Carlo Tree Search can be used to schedule intentions at the action level so as to avoid negative interactions.

The approaches above make use of information derived from the pre- and postconditions of actions and the context conditions of plan-rules. An alternative approach is for developers to provide additional information about plan-rules. For example, Bordini et al. [2002] proposed an extended version of AgentSpeak, AgentSpeak(XL), that uses a scheduler to generate schedules for AgentSpeak intentions, e.g., to decide which goals to perform, how to perform them, and the order in which they should be performed. Their approach relies on the manual provision of relations between plan-rules, e.g., whether a plan 'facilitates' or 'hinders' some other plan.

6.3 Meta Level Reasoning

In many languages, the choice between the applicable plan-rules for a goal is arbitrary or based on the order in which rules appear in the agent program, and changing this default behaviour requires essentially re-implementing part of the interpreter. In addition to failure handling as discussed above, meta-level reasoning can be used to choose the most appropriate applicable plan-rule for a goal. One approach is for meta-level plan-rules to have higher priority than standard plan-rules (as in OpenPRS and meta-APL [Leask and Logan, 2018]). Alternatively, the existence of more than one applicable plan-rule for a goal triggers a special goal that is handled by a meta-level plan-rule (as in JACK). This is used to reason about the computed set of applicable plan-rules (e.g., by reading a 'meta-predicate' which stores this set) and then select a plan-rule from this set for the current situation, based on criteria such as the success rate of executing that rule in the past. Meta-level reasoning can also be used to access and modify values in the agent's internal data structures (e.g., using 'meta-functions' as in Jason), in order to, for example, abort a declarative goal being pursued. Finally, meta-level reasoning can be used in OpenPRS to execute all the agent's intentions (truly) concurrently, rather than interleaving them.

7 Discussion

In this section, we identify some key open research questions relating to beliefs, goals, plans and the deliberation cycle, and highlight some possible directions for future work in BDI agent architectures.

While the simple approach to beliefs currently used by many agent programming languages is sufficient for a large class of applications, it seems likely that richer representations will be required for emerging application areas such as 'assistive agents' that must maintain models of the beliefs and goals of the humans they are assisting, and robotics and autonomous systems which must integrate sub-symbolic information from sensors into the agent's beliefs. This is likely to lead to work in, e.g., representing nested and uncertain beliefs, and belief revision among others.

The 'long term autonomy' necessary for many autonomous systems will also drive future research on approaches to goals. For example, it is likely that reasoning about maintenance goals and their interaction with achievement goals will become more important, as will properties and constraints on goals such as priorities and deadlines. Previous work in this area, e.g., [Harland *et al.*, 2014] requires the developer to anticipate and manage possible interactions between goals, which is likely to be infeasible for agents with extended 'lifetimes'. In addition, most BDI languages provide no support for deliberating about whether an agent should adopt a goal, e.g., an autonomously generated goal. Future work in this area may draw on recent work on goal reasoning [Coman and Aha, 2018] that attempts to provide mechanisms for goal generation and management.

Robotics applications are also likely to place greater emphasis on truly concurrent plan execution, leading to work on, e.g., formalising interactions between concurrently executing branches, and between the agent's deliberation cycle and durative actions. Another area of future work involves the use of AI techniques such as machine learning to supplement pre-defined plan-rules or generate new rules. Previous work in this area, e.g., [Singh and Hindriks, 2013], has focussed on learning context conditions for plan-rules, but it would be interesting to explore the integration of learnt behaviours into a BDI framework (perhaps using techniques from the work on learning HTN methods), and to extend previous work on the generation of new plans at runtime using AI planning [Meneguzzi *et al.*, 2015].

The application areas highlighted above also have implications for future work on the agent's deliberation cycle. For example, support for uncertain beliefs has implications both for when a declarative goal may be considered 'achieved' (or unachievable), and for intention scheduling approaches based on simulations of interleavings, as in [Yao and Logan, 2016]. Similarly, we would expect to see further work on other aspects of intention scheduling, such as deadlines, resource usage, and compliance with social norms [Meneguzzi *et al.*, 2012] that are critical in many autonomous systems.

In many cases, the research questions identified above draw on synergies with other areas of AI, e.g., machine learning, HTN planning, and NLP; we believe this will be an important direction for future research in BDI agent architectures.

Acknowledgements

Felipe Meneguzzi acknowledges support from CNPq under project numbers 407058/2018-4 and 302773/2019-3.

References

[Alechina *et al.*, 2006] Natasha Alechina, Rafael Bordini, Jomi Hübner, Mark Jago, and Brian Logan. Automating belief revision for AgentSpeak. In *Proceedings of the 4th International Workshop on Declarative Agent Languages* *and Technologies*, volume 4327 of *Lecture Notes in Computer Science*, pages 61–77, 2006.

[Bagosi *et al.*, 2015] Timea Bagosi, Joachim de Greeff, Koen V. Hindriks, and Mark A. Neerincx. Designing a knowledge representation interface for cognitive agents. In *Engineering Multi-Agent Systems*, pages 33–50, 2015.

[Bauters *et al.*, 2014] Kim Bauters, Weiru Liu, Jun Hong, Carles Sierra, and Lluis Godo. CAN(PLAN)+: extending the operational semantics of the BDI architecture to deal with uncertain information. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 52–61, 2014.

[Bordini *et al.*, 2002] Rafael H. Bordini, Ana L. C. Bazzan, Rafael de O. Jannone, Daniel M. Basso, Rosa M. Vicari, and Victor R. Lesser. AgentSpeak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1294–1302, 2002.

[Bordini *et al.*, 2006] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah-Seghrouchni, Jorge J. Gómez-Sanz, João Leite, Gregory M. P. O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

[Bordini *et al.*, 2007] Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[Bratman, 1987] Michael E. Bratman. *Intention, Plans and Practical Reason*. Harvard University Press, 1987.

[Braubach *et al.*, 2005] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Jadex: A BDI-agent system combining middleware and reasoning. In Rainer Unland, Monique Calisti, and Matthias Klusch, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168, 2005.

[Busetta *et al.*, 1999] Paolo Busetta, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. JACK intelligent agents - components for intelligent agents in Java. AgentLink Newsletter, 1999. White paper, http://www.agent-software.com.au.

[Casali *et al.*, 2011] Ana Casali, Lluis Godo, and Carles Sierra. A graded BDI agent model to represent and reason about preferences. *Artificial Intelligence*, 175(7-8):1468–1478, 2011.

[Chaouche *et al.*, 2014] Ahmed Chawki Chaouche, A El Fallah Seghrouchni, Jean-Michel Ilié, and Djamel-Eddine Saïdouni. A dynamical plan revising for ambient systems. *Procedia Computer Science*, 32:37–44, 2014.

[Coelho and Nogueira, 2015] Francisco Coelho and Vítor Nogueira. Probabilistic perception revision in AgentSpeak(L). In *Proceedings of the International Conference on Principles and Practice of Multi-Agent Systems*, volume 9387 of *Lecture Notes in Computer Science*, pages 613–621, 2015.

[Coman and Aha, 2018] Alexandra Coman and David W Aha. AI rebel agents. *AI Magazine*, 39(3):16–26, 2018.

[Dastani *et al.*, 2004] Mehdi Dastani, M. Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer. A programming language for cognitive agents: Goal directed 3APL. In *Programming Multi-Agent Systems*, pages 111–130, 2004.

[Dastani *et al.*, 2006] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Goal types in agent programming. In *Proceedings of the European Conference on Artificial Intelligence*, pages 220–224, 2006.

[Dastani *et al.*, 2009] Mehdi M. Dastani, Koen V. Hindriks, Peter Novák, and Nick A. M. Tinnemeier. Combining multiple knowledge representation technologies into agent programming languages. In *Declarative Agent Languages and Technologies VI*, pages 60–74, 2009.

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

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

[de Silva *et al.*, 2018] Lavindra de Silva, Felipe Meneguzzi, and Brian Logan. An operational semantics for a fragment of PRS. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 195–202, 2018.

[de Silva, 2018] Lavindra de Silva. HTN acting: A formalism and an algorithm. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 363–371, 2018.

[de Silva, 2020] Lavindra de Silva. An operational semantics for true concurrency in BDI agent systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[Georgeff and Lansky, 1987] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 677–682, 1987.

[Ghallab *et al.*, 2016] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

[Harland *et al.*, 2014] James Harland, David N. Morley, John Thangarajah, and Neil Yorke-Smith. An operational semantics for the goal life-cycle in BDI agents. *Autonomous Agents and Multi-Agent Systems*, 28(4):682–719, 2014.

[Hindriks *et al.*, 1999] Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[Hindriks *et al.*, 2001] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. In *Proceedings of* *the 7th International Workshop on Agent Theories Architectures and Languages*, volume 1986 of *Lecture Notes in Computer Science*, pages 228–243. 2001.

[Hindriks *et al.*, 2009] Koen V. Hindriks, Wiebe van der Hoek, and M. Birna van Riemsdijk. Agent programming with temporally extended goals. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 137–144, 2009.

[Huber, 2001] M. J. Huber. JAM agents in a nutshell, version 0.61+0.79i. http://www.marcush.net/irs/jam/jam-man-01nov01-draft.htm, November 2001.

[Hübner *et al.*, 2006] Jomi Fred Hübner, Rafael H. Bordini, and Michael Wooldridge. Programming declarative goals using plan patterns. In *Proceedings of the 4th International Workshop on Declarative Agent Languages and Technologies*, volume 4327 of *Lecture Notes in Computer Science*, pages 123–140, 2006.

[Ingrand, 2014] Félix Ingrand. *OPRS Development Environment*, version 1.1b7 edition, 2014.

[Klapiscak and Bordini, 2009] Thomas Klapiscak and Rafael H. Bordini. JASDL: A practical programming approach combining agent and semantic web technologies. In *Declarative Agent Languages and Technologies VI*, pages 91–110, 2009.

[Kravari and Bassiliades, 2015] Kalliopi Kravari and Nick Bassiliades. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 8(1):11, 2015.

[Kwisthout and Dastani, 2005] Johan Kwisthout and Mehdi Dastani. Modelling uncertainty in agent programming. In *Declarative Agent Languages and Technologies III*, pages 17–32, 2005.

[Leask and Logan, 2018] Sam Leask and Brian Logan. Programming agent deliberation using procedural reflection. *Fundamenta Informaticae*, 158(1–3):93–120, 2018.

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

[Meneguzzi *et al.*, 2012] Felipe Meneguzzi, Sanjay Modgil, Nir Oren, Simon Miles, Michael Luck, and Noura Faci. Applying electronic contracting to the aerospace aftercare domain. *Engineering Applications of Artificial Intelligence*, 25(7):1471–1487, 2012.

[Meneguzzi *et al.*, 2015] Felipe Meneguzzi, Odinaldo Rodrigues, Nir Oren, Wamberto W. Vasconcelos, and Michael Luck. BDI reasoning with normative considerations. *Engineering Applications of Artificial Intelligence*, 43(0):127–146, 2015.

[Moreira *et al.*, 2006] Álvaro F. Moreira, Renata Vieira, Rafael H. Bordini, and Jomi F. Hübner. Agent-oriented programming with underlying ontological reasoning. In *Proceedings of the 3rd Internatinal Workshop on Declarative Agent Languages and Technologies*, pages 155–170, 2006.

[Panisson *et al.*, 2018] Alison R. Panisson, Simon Parsons, Peter McBurney, and Rafael H. Bordini. Choosing appropriate arguments from trustworthy sources. In *Computational Models of Argument*, pages 345–352, 2018.

[Rao, 1996] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proceedings of the European workshop on Modelling Autonomous Agents in a Multi-Agent World : Agents Breaking Away*, pages 42–55, 1996.

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

[Silva and Gluz, 2011] Diego Goncalves Silva and Joao Carlos Gluz. AgentSpeak(PL): A new programming language for BDI agents with integrated Bayesian network model. In *Proceedings of the International Conference on Information Science and Applications*, pages 1–7, 2011.

[Singh and Hindriks, 2013] Dhirendra Singh and Koen V. Hindriks. Learning to improve agent behaviours in GOAL. In *Programming Multi-Agent Systems*, volume 7837 of *Lecture Notes in Computer Science*, pages 158–173. 2013.

[Thangarajah and Padgham, 2011] John Thangarajah and Lin Padgham. Computationally effective reasoning about goal interactions. *Journal of Automated Reasoning*, 47(1):17–56, 2011.

[Vikhorev *et al.*, 2011] Konstantin Vikhorev, Natasha Alechina, and Brian Logan. Agent programming with priorities and deadlines. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 397–404, 2011.

[Waters *et al.*, 2014] Max Waters, Lin Padgham, and Sebastian Sardiña. Evaluating coverage based intention selection. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems*, pages 957–964, 2014.

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

[Wobcke, 2001] Wayne Wobcke. An operational semantics for a PRS-like agent architecture. In *Proceedings of the Australian Joint Conference on Artificial Intelligence*, pages 569–580, 2001.

[Yao and Logan, 2016] Yuan Yao and Brian Logan. Action-level intention selection for BDI agents. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 1227–1236, 2016.

[Yao *et al.*, 2016] Yuan Yao, Brian Logan, and John Thangarajah. Robust execution of BDI agent programs by exploiting synergies between intentions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2558–2565, 2016.