Encapsulating Reactive Behaviour in Goal-based Plans for Programming BDI Agents

Paper #1075

ABSTRACT

Reactive behaviour in BDI-based models and architectures adopted in agent programming is typically specified in terms of reactive plans not bound to any specific goal. In this paper, we present and discuss an extension of the plan model used in BDI programming languages in which goal-based plans encapsulate both proactive and reactive behaviour. This brings important benefits both to the practice of agent programming and in supporting agent reasoning at runtime. The paper first introduces the model formally, abstracting away from particular programming languages. The approach is then evaluated through concrete implementations based on two existing agent programming platforms, namely Jason and ASTRA. The paper reports on experiments showing that the approach provides for elegant programming and can also be efficient.

KEYWORDS

proactive and reactive behaviour; agent programming; BDI agents

1 INTRODUCTION

The BDI architecture and model has been either the basis or at least an important influence in many agent programming languages. This has led to many concrete computational models and platform implementations, even if the basic programming language is a traditional one, for example PRS [17], dMARS [11], JAM [16], JACK [32], and SPARK [20], to name just a few. It also leads to the creation of various programming languages and variants of those languages, whether abstract (and used mostly for formalisation of ideas related to BDI) or more practical, for example AgentSpeak(L) [25], CAN [28], Jason [1], ASTRA [7], and Gwendolen [10], again to name just a few.

Needless to say, plans have a key role in programming BDI agents, regardless of the particular approach. Plans define the agent know-how, they express a course of action that can be used to bring about a state-of-affairs, particularly those that are desirable to the agent, hence plans have a strong relation to goal-orientation. A plan has either a goal to achieve or to maintain and so they form an essential part of agents' behaviour.

However, the plan model adopted in all concrete BDI-based computational platforms, from the very early days, have pitfalls that long-term experience has helped surface. One of them is a lack of structure with the plan library, and in particular lack of encapsulation. A plan library is usually just a collection of plans to achieve individual (top-level) goals and their subgoals. However, the intentional context in which a subgoal needs to be pursued is relevant, that is, in many cases the agent has few top-level goals, and subgoals might need to be pursed differently depending on which top-level goal has originated the subgoal. Another typical problem is that, because agents need to be reactive as well as proactive, it has been tempting to just use a plan programming construct to specify reactive behaviour, but again these may vary significantly depending on the context of the overall top-level goals that the agent is currently trying to achieve.

This paper puts forward an approach to address these problems. It preserves goal-orientation while being cleanly event-driven. Most importantly, it increases encapsulation in defining the strategies for plans to achieve/maintain goals. This not only has a clear impact both at the design and programming activities, but also for reasoning about gals and intentions at runtime. In particular, we conjecture it may have an impact in the intention progression problem [18].

We first present our extended plan model formally and in a language-independent way. Then, we present implementations of our approach extending two different existing BDI platforms, namely Jason and ASTRA. We show experimental results to assess how the extended platforms compare to the original ones. We then discuss how this approach addresses the pitfalls of traditional BDI plan models.

2 BACKGROUND AND MOTIVATION

The Belief-Desire-Intention (BDI) architecture is one of the major models adopted in academia and industry for developing intelligent agents and in particular practical reasoning agents. One of the first concrete architectures based on BDI was PRS [17], implemented then by the dMARS system [11] and applied in several significant multi-agent applications. A whole range of practical development efforts relating to BDI systems have been undertaken, either as refinements of PRS and dMARS or more loosely based on broader BDI principles, including PRS-Lite, JAM, JACK, Jadex. Besides systems, several agent programming languages have been developed based on BDI (surveys can be found here [2, 3]). Among them, we mention here AgentSpeak(L) [25] — an abstract language based on an abstraction of the PRS architecture, stripped down to its bare essentials, later extended and implemented by concrete Agent Programming platforms such as Jason and ASTRA — and influencing many languages including the more recent CANPlan [29], which is equipped with a formal specification.

In spite of specific differences, all BDI computational models are based on the same conceptual model for plans and intentions. A *plan* represents how to bring about a state of affairs. *Intentions* represent a specific state of affairs that the agent is committed to achieving and the activity used for that purpose. A plan then is a recipe specifying the course of action that may be undertaken by an agent in order to achieve such states of affairs. Plans are collected

Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), B. An, N. Yorke-Smith, A. El Fallah Seghrouchni, G. Sukthankar (eds.), May 2020, Auckland, New Zealand

https://doi.org/doi

in agent's plan library, representing its *procedural knowledge* or *know-how*.

The issues that we focus on this paper are about how this conceptual model is then reified in concrete computational and programming models. Each plan is typically defined by some key components, including [11]:

- a *trigger* or *invocation* condition, which specifies the circumstances under which the plan should be considered relevant, usually specified in terms of *events*;
- a *context*, or *pre-condition*, specifying the circumstances under which the execution of the plan may commence and the plan considered applicable;
- a *body*, defining a potentially quite complex course of actions which may consist of both goals (or subgoals) and primitive actions.

This model is pervasively used, from the original dMARS specification up to the more recent CANPlan [29].

By exploiting the model in practice in AOP, two main relevant issues emerged, as described below.

Plans without explicit goals. The invocation condition could be both events concerning new goals or belief changes related to percepts from the environment or data in general. The latter case makes it possible to define reactive behaviours and data-driven / event-driven processing. For example, the plan "makeTea" may be triggered by the event "thirsty" [11].

By adopting this choice in practice, a couple of important characteristics can be noted. First, in plans triggered by invocation conditions that concern environment/data events, the state of affairs remains implicit, in the mind of the designer. This has a drawback at runtime: intentions that are created to execute the plan are taskless/goal-less, i.e., they do not have an explicit "state of affairs" they are associated with. From a design/engineering point of view, the reactive behaviour of an agent is always motivated by some task to be achieved, a "state of affairs" to be achieved. In the example, the goal is to make a tea, to have a new tea. A robot that reacts to a low-battery charge event and goes back to the recharge station does so due to a maintenance task that could be described as "keep the battery level not lower than some threshold".

Second, this choice impacts strongly on modularity and reuse. From the example, there could be multiple reasons for making a tea, not only being thirsty, and the plan – as a recipe for making tea – would be the same. However, if we put the triggering condition in the plan, that plan cannot be reused for different triggering conditions. In this case the invocation condition appears better modelled as the motivation for adopting some specific goal ("to make a cup of tea") and we can have different plans for it depending on the context. But being thirsty in this case would not be part the plan itself.

Reactive behaviour in plan strategy. The plan body is meant to represent a recipe for how to achieve some state of affairs. In many relevant cases in practice, such a recipe may include the capability to asynchronously react to events from the environment. In some cases this is about critical situations – e.g., low battery for a robot cleaning the floor. In some other cases, it could be strongly related to the designed strategy to fulfil the task – e.g., reacting to messages received or monitoring some state of the ongoing work done in the environment. In the general case, the strategy used to achieve the state of affairs may be required to flexibly mix proactive and reactive behaviour, but in the context of the same intention.

The model for plans (body) in BDI — i.e., a sequence of actions and (sub-)goals — does not make this straightforward. If we need to react to some event e in the context of a plan to achieve some goal G, then a different plan specifying e as triggering condition must be used. The effect is a poor level of *encapsulation* about the plan strategy, which must be necessarily specified in terms of a set of unrelated plans. As in the previous case, the relation between the plans is in the mind of the designer, but is neither expressed explicitly in the source code nor is it captured by intentions at runtime.

To deal with these issues, we propose an extension of the plan model adopted in BDI agents which:

- enforces goal/task orientation, that is: every plan p has an explicit account for the task t to be either achieved or maintained. The unique invoking condition is always a goal/task to be achieved or maintained. This implies that every intention at runtime – as a plan in execution – is bound to an explicit goal.
- extends the plan specification to include both subplans and reactive behaviour, besides a plan body having squences of actions and subgoals, so as to get full encapsulation of proactivity and reactivity in the definition of the strategy of a plan.

3 PROPOSAL

In this section, our proposal is presented formally and independent of a particular programming language. This makes our approach more easily applicable to the various existing agent programming languages. Later in this paper we show how this has been implemented in two different agent platforms.

The presentation is formal yet language independent with the help of an abstract syntax for relevant structures such as plans, intentions, events, and so forth. The grammar below formally defines sets of such structures which can then be formally used in an algorithm of how a general BDI interpreter should be adapted to follow our approach. However, because there is no concrete syntax associated with those sets BDI structures, this makes any BDI-inspired platform that has particular representations for those structures able to use the approach in a straightforward manner. The abstract syntax is shown in Figure 1.

$$
\begin{array}{lcll}
ag & ::= & bs\ gs\ gps & \\
bs & ::= & b_1 \ldots b_n & (n \geq 0) \\
gs & ::= & g_1 \ldots g_n & (n \geq 0) \\
ps & ::= & p_1 \ldots p_n & (n \geq 1) \\
p & ::= & g\ f_c\ f_g\ h\ pr_1 \ldots pr_n & (n \geq 0) \\
r & ::= & (+b \mid -b)\ f\ h & \\
pr & ::= & p \mid r & \\
t & ::= & g \mid +b \mid -b & \\
h & ::= & d_1 \ldots d_n & (n \geq 0) \\
d & ::= & a \mid g &
\end{array}
$$

Figure 1: Abstract Syntax

The grammar in Figure 1 defines all structures of interest to our BDI approach, where b is a metavariable standing for an individual belief an agent may have, g a goal, a is an action, and f a logical formula. Note that how these and other data structures required in our presentation here are effectively expressed in a concrete programming platform is irrelevant from the abstract syntax point of view and therefore easy to adapt to whatever notion of such structures that a particular BDI programming language platform has.

Overall the grammar states that an agent is defined as a set of (initial) beliefs, a set of (initial) goals, and a set of plans. Each plan $p_i \in ps$ is a tuple that has an event, a formula often referred to as the *context* or *guard* for the plan (a formula checked when we are selecting a plan for an event), another formula that in [26] has been named the *goal condition* (the goal is achieved when/if this formula becomes true given the agents beliefs). Plans p and reactive rules r are triggered (t) by a goal or changes (addition '+' or deletion '−') in beliefs, respectively. The body (h) of plans and rules might be empty but otherwise is sequence of deeds (d), a name used in [10] to refer to either goals to achieve or actions to execute. The deeds in a plan body start to be executed when an instance of that plan/rule becomes an intention. Within a plan there is a finite sequence of other such plans or reactive rules (pr).

We now show some algorithms that completely define our proposal in a general way for any BDI interpreter. Algorithm 1 simply sets up the agent's initial state, with the help of the function for generating events in Algorithm 2, and all the important operations of our approach is given as an agent reasoning cycle as shown in Algorithm 3. That algorithm makes reference to another algorithm used to retrieve relevant plans, shown as Algorithm 4, which in our approach is different than in previous agent languages, because the plans give a context in which to look for relevant plans and reactive rules.

Note that the non-terminals of the grammar define syntactic categories, for example bs is a set of beliefs, and when needed we will index its elements by natural numbers. For example, given an agent $ag = \langle bs, gs, ps \rangle$, we may refer to $b_i \in bs$ as the agent's i-th initial belief. In the algorithms introduced below, if a variable name coincides with a syntactical category of the grammar in Figure 1, that clearly defines its type. Otherwise, we use the syntactic categories to express the type of a variable (or indeed to check whether the content of a variable) using the notation $u : c$ to say that the content of variable u conforms to the syntactic category c. So, for example, if we say $B : bs$ we mean that variable B contains a set of beliefs.

Besides the data structures defined in the grammar, we need structures for events and intentions. An intention i is a stack of plans, each with its own stack of deeds (those that are internal to plan or coming from activated reactive rules), denoted as $[p_1[d_{1_1}, \ldots, d_{1_{n1}}], \ldots, p_n[d_{n_1}, \ldots, d_{n_{nk}}]]$ where p_1 is the plan at the top of that intention stack and d_{1_1} is the deed at the top of p_1's stack of deeds (and therefore the next deed to be executed). Events typically refer to changes in beliefs or goals the agent has adopted, so an event e is a tuple $\langle t, i \rangle$ where t is a trigger (as defined in the grammar above) and i an intention, possibly [], the empty intention. As in some agent languages, when an event is associated with an empty intention we call it *external*; external events arise from perception of the environment, agent communication (for example, goal delegation) or initial goals, whereas internal events are associated with an intention (for example, a plan being executed requires the achievement of a subgoal).

The main agent algorithm is shown in Algorithm 1, which simply initialises variable B for the agent beliefs, P for the agent plans, and the set of events E initially contains one event for each initial goal in the program (if any). The set of intentions I is initially empty. In this presentation an agent enters an infinite loop within which it: (i) perceives the environment with a function CURRENT_PERCEPTS which is assumed as given (i.e., a concrete agent architecture will have the ability to do so); (ii) generate events for all *changes* in beliefs caused by the received percepts (see Algorithm 2); (iii) uses the REASONING_CYCLE function shown in Algorithm 3 to decide on the next action to take; an (iv) executes that action (again it is assumed that the actual agent architecture provides the means for executing an action, which expressed in the algorithm by function EXECUTE).

Algorithm 1 Agent Initialisation

Require: an initial agent program $ag = \langle bs, gs, ps \rangle$

```
1:  B ← bs
2:  E ← {}
3:  for all g ∈ gs do
4:      E ← E ∪ {⟨g, []⟩}
5:  P ← ps
6:  I ← {}
7:  while true do
8:      S ← CURRENT_PERCEPTS
9:      GENERATE_EVENTS(S)
10:     action ← REASONING_CYCLE( )
11:     EXECUTE(action)
```

Algorithm 2 simply checks whether there are received percepts (i.e, symbolic information saying what is currently perceived as true in the environment) which are not currently in the belief base and generate external events for the addition of those beliefs (line 6). It then checks if there are beliefs which are no longer perceived as true in the environment and generates belief deletions events for those (line 10). The belief base itself is updating accordingly.

Algorithm 2 Event Generation from Percepts

Require: external variables B, E

```
1:  function GENERATE_EVENTS(S)
2:      ▷ S is a set of percepts from the agent's sensors
3:      for all s ∈ S do
4:          if s ∉ B then
5:              B ← B ∪ {s}
6:              E ← E ∪ {⟨+s, []⟩}
7:      for all b ∈ B do
8:          if b ∉ S then
9:              B ← B \ {b}
10:             E ← E ∪ {⟨−b, []⟩}
```

The main algorithm is in fact Algorithm 3, which shows the reasoning cycle for a BDI agent following our approach. Recall

that in our approach we have a "goal condition" which, when true, implies that a corresponding intended means can be dropped. This leads to some computational burden but it is an important feature as it ensures the agent will not take unnecessary action. This is done in lines 3–8, starting from the bottom of the intention because dropping a goal near the bottom, if possible, will eliminate all the stack of plans on top of it within the intention.

We then handle one event from the set of events E. Note that, as in AgentSpeak, we assume there are user-defined functions to select an event, an option or intended means (one among possibly various applicable plans), and an intention from the set of intentions to execute next, respectively called in the algorithm SELECT_EV, SELECT_OPT, and SELECT_INT. Function GET_APPLICABLE is not given an algorithm because it simply checks if the context part of the plans or rules are true (depending on the particular language, this might include checking for logical consequence from the belief base, for example). After one event is selected, there are different parts of the algorithm depending on what type of event was selected.

For a belief-change event, this is handled in lines 13–21. Differently from other BDI platforms, our approach will replicate the reactive event to each existing intention for which it is relevant. So for each intention in the set of intention, we search for relevant plans in the entire intention, which in our notation here includes syntactical copies of the plan being executed, including therefore the plans encapsulated in them. If there are relevant and applicable plans for that particular intention, one of those is selected and its body executed. Again, we emphasise this process is repeated for each single intention.

External goal events, i.e., initial goals or goals delegated by other agents, are handled in lines 23–29. For those, after selecting one applicable relevant plan, we simply create a new intention for it if the goal condition is not yet believed true; the intention is a stack with a single element which has a copy of the plan p and a list of deeds to be executed (the body of that plan).

Internal goal events, i.e., subgoals that appear in currently executing plans, are handled in lines 31–38. This is very similar to the case above except that new the intended means is pushed on top of the existing intention.

The final part of the algorithm selects one intention to be further executed in that reasoning cycle. If the first deed not yet executed in the body of the topmost plan in the intention is a goal, the appropriate event is generated and the reasoning cycle function called recursively until a next action to execute is determined. If the deed is an action, the intention is updated to reflect that the action is going to be executed (i.e., it is removed of the list of deeds to be executed) and the action is simply returned.

Note that in Algorithm 3 we do not give the details of what happens when a stack of deeds associated with a plan in the intention stack becomes empty. In practice that leads to the removal of that plan from the intention stack and possibly the subgoal in the list of deeds of the plan below it in the intention stack (or the entire intention if the list of deeds at the bottom is then empty).

Finally, Algorithm 4 shows a recursive function which receives as parameter a trigger t to be matched with the triggers of plans anywhere in the intention i and also in top-level goals of the plan library (note that in our approach, reactive rules do not appear at the top-level of a plan library, only associated with the goals

Algorithm 3 Reasoning Cycle

Require: external variables B, E, P, I

1: **function** REASONING_CYCLE
2: ▷ Drop all achieved goals
3: **for all** $i \in I$ **do**
4: **for** j = LENGTH(i) down to 1 **do** ▷ From bottom to top
5: $i'' \leftarrow [p_j[h_j]] \in i$
6: let $p_j = \langle t, f_c, f_g, h, pr_1, \ldots, pr_n \rangle$
7: **if** $B \models f_g$ **then**
8: $I \leftarrow I \setminus \{i\} \cup \{\text{REMOVE}(i'', i)\}$
9: ▷ Handle an event
10: $se \leftarrow$ SELECT_EV(E)
11: let $se = \langle t, i \rangle$
12: **if** $i = \{\} \wedge t : (+b \mid -b)$ **then** ▷ external belief event
13: **for all** $i \in I$ **do** ▷ may trigger a r.r. in each intention
14: rps = GET_RELEVANT($t, i, \{\}$) ▷ search entire i
15: aps = GET_APPLICABLE(rps)
16: **if** $aps \neq \{\}$ **then**
17: r = SELECT_OPT(aps)
18: let $r = \langle t f h \rangle$
19: let $i = [p_1[h_1], i']$
20: $i'' \leftarrow [p_1[h, h_1], i']$
21: $I \leftarrow I \setminus \{i\} \cup \{i''\}$
22: **else if** $i = \{\} \wedge t : g$ **then** ▷ external goal event
23: $rps \leftarrow$ GET_RELEVANT($t, [], \{\}$) ▷ search only in P
24: $aps \leftarrow$ GET_APPLICABLE(rps)
25: **if** $aps \neq \{\}$ **then**
26: $p \leftarrow$ SELECT_OPT(aps)
27: let $p = \langle t, f_c, f_g, h, pr_1, \ldots, pr_n \rangle$
28: **if** $B \not\models f_g$ **then**
29: $I \leftarrow I \cup \{[p[h]]\}$
30: **else if** $i \neq \{\} \wedge t : g$ **then** ▷ internal (goal) event
31: $rps \leftarrow$ GET_RELEVANT($t, i, \{\}$) ▷ search in i
32: ▷ and top level plans in P
33: $aps \leftarrow$ GET_APPLICABLE(rps)
34: **if** $aps \neq \{\}$ **then**
35: $p \leftarrow$ SELECT_OPT(aps)
36: let $p = \langle t, f_c, f_g, h, pr_1, \ldots, pr_n \rangle$
37: **if** $B \not\models f_g$ **then**
38: $I \leftarrow I \setminus \{i\} \cup \{\text{PUSH}(p[h], i)\}$
39: ▷ Execute a step of an intention
40: $i \leftarrow$ SELECT_INT(I)
41: let $i = [p_1[d_1, h_1], i']$
42: **if** $d_1 : g$ **then**
43: $E \leftarrow E \cup \{\langle d_1, i \rangle\}$
44: $action \leftarrow$ REASONING_CYCLE( )
45: **return** $action$
46: **else if** $d_1 : a$ **then**
47: $i'' \leftarrow [p_1[h_1], i']$
48: $I \leftarrow I \setminus \{i\} \cup \{i''\}$
49: **return** d_1
50: ▷ See note in the text about empty stacks

Algorithm 4 Retrieving Relevant Plans

Require: external variable P

```
1:  function GET_RELEVANT(t,i,rps)
2:      if i ≠ {} then
3:          let p[h] = HEAD(i)
4:          for all pr ∈ p do
5:              if RELEVANT(pr,t) then
6:                  rps ← rps ∪ {pr}
            GET_RELEVANT(t, TAIL(i), rps)
7:      else
8:          for all p ∈ P do    ▷ check for relevant top-level plans
9:              if RELEVANT(p,t) then
10:                 rps ← rps ∪ {p}
11:     return rps
```

that the agent may need to achieve). The final parameter is a set of plans already determined to be relevant for t, normally empty when the function is initially called. Function RELEVANT simply tries to match the triggering-event of plan p with t (the implementation of this function is of course language dependent; it might require for example unification if the language is logic based).

4 IMPLEMENTATION AND EVALUATION

To assess the computational viability and characteristics of the proposed language, we extended the interpreters of both ASTRA [7] and Jason [4], called ASTRA(ER) and Jason(ER) respectively, with the new features discussed in this paper. The implementation has helped us not only confirm that the ideas are feasible, but provided refinements for the model, as well as being a tool to measure how the approach scales and compares against other languages.[1]

4.1 Jason(ER)

An example of a concrete program in Jason(ER) is shown in Listing 1. It implements plans for an initiator agent in the context of the Contract Net Protocols (CNP) [12]. The main part of the program is a plan to achieve the goal of running a CNP for some given task (lines 4–14). This plan encapsulates a body (line 5) and three sub-plans (lines 6–13). The body has three sub-goals: announce the call for proposals (CFP), wait for bids, and contract the winner. We highlight here the sub-plan for the sub-goal `bids`. It is considered achieved when either (*i*) all participants have sent an answer (a proposal or a refusal) or (*ii*) a deadline has passed (four seconds in this case). The rule on line 1 is used to evaluate condition (*i*) and the `.wait` on line 7 is used for condition (*ii*).

The challenge is on the interplay of those two conditions. The solution here uses two features proposed in this paper: goal conditions and sub-plans and reactive rules. The goal condition is placed after `<:` on line 6 and is `false`. Since this condition is never satisfied in the mental state of the agent, the internal action `.done` is used to finish the goal. The reactive rules on lines 9 and 10 are used to react to the answers. Thus, while running line 7 of the plan body, for every received answer, one of those two reactive rules is selected and, if enough answers have being received, it finishes the goal `bids`. If none of these two rules are executed, the plan body

[1] ASTRA(ER) is available at xxxxxxxx and Jason(ER) is available at xxxxxxxx.

Listing 1 Jason(ER) implementation of CNP initiator

```
1  all_ans(I) :- ... // true if all participants have answered
2
3  // plan to achieve goal cnp, I identifies the CNP
4  +!cnp(I,Task) {
5      <- announce_cfp(I,Task); !bids(I); !contract(I).
6      +!bids(I) <: false {
7          <- .wait(4000); .done.
8          // reaction to the event of new proposal / refusal
9          +propose(I,_) : all_ans(I) <- .done.
10         +refuse(I)    : all_ans(I) <- .done.
11     }
12     +!announce_cfp(I,Task) <- ...
13     +!contract(I) <- ...
14 }
```

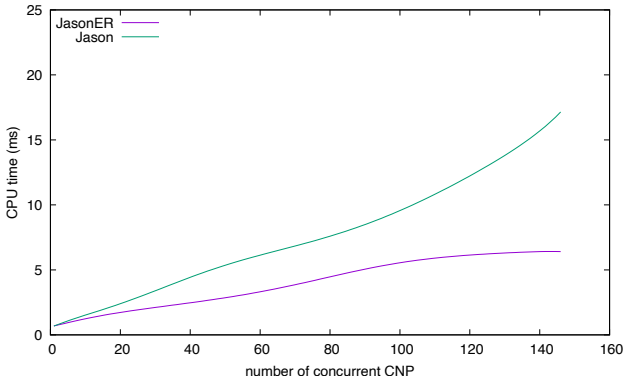

Figure 2: Jason(ER) performance evaluation.

executes `.done` after four seconds, continuing with the sequence on line 7.

Considering this program, we emphasise that:

- The plan to achieve goal `bids` *encapsulates* both proactive and reactive behaviour. The former as a sequence of actions (the body) and the latter as a set of (encapsulated) reactive rules.
- The reaction to answers is defined in the context of the goals `bids` and `cnp`. An agent knows therefore why (for which goal) it is executing those reactive rules.

The implementation of Jason(ER) allowed us to evaluated how it scales considering a MAS that concurrently runs n CNPs. It is expected that the time required to finish n CNPs increases linearly on n. The MAS has one agent playing initiator and eleven playing participant. Only the initiator uses the new features of Jason(ER) as shown in Listing 1. The result of the experiment, shown in Figure 2, confirms that Jason(ER) scales linearly on the number of CNPs.[2]

In a second experiment, we intend to evaluate the overhead of the new features. We thus run the same MAS replacing the initiator agent by one implemented as usual in Jason (its program is shown in Listing 2). We noted that Jason(ER) is indeed faster than Jason for

[2] The code and data required to repeat the experiment, as well as more details, are available at xxxxxxxx.

Listing 2 Jason implementation of CNP initiator

```
1  +!cnp(I,Task) <- announce_cfp(I,Task); !bids(I).
2
3  +!bids(I) <- .at("now +4 seconds", { +!contract(I) }).
4
5  +propose(I,_) : all_ans(I) <- !contract(I).
6  +refuse(I)    : all_ans(I) <- !contract(I).
7
8  +!announce_cfp(I,Task) <- ...
9  +!contract(I) : not .intend(contract(I)) <- ...
```

this application (cf. Figure 2). The reasons for this old Jason agent being slower are the following:

(1) While sending the CFP and before finishing this task, the agent starts receiving proposals and refusals for the first sent messages. This agent has relevant plans for these events (lines 5–6) even if (for sure) these plans are not applicable yet — they can be applicable only after the last CFP was sent. The agent is thus wasting time considering the plans on lines 5 and 6 before finishing the CFP announcements.
(2) The contract goal (line 9) needs to be singleton for a CNP (we cannot contract twice for the same CNP), however plans on lines 5 and 6 may trigger this goal more than once. Thus the context of this plan tests if no other contract intention is running for the same CNP; the internal action `.intend` performs this test. This internal action complexity is $O(m)$ (where m is the number of active intentions). Jason(ER) avoids these two issues: plans for proposals and refusals are relevant only when the goal `bids` is being pursued (neither before, nor after); the goal `contract` is pursued after the goal `bids` and not as a consequence of a reaction.

Despite the improvements in the performance (that could be circumstantial for this very experiment/application), the main advantage of Jason(ER) is how long-term goals (line 5 in Listing 1) are explicitly related to relevant reactions (lines 9–10). The decomposition into sub-goals is also more explicit, as we can see when comparing line 5 in Listing 1 against line 1 in Listing 2. While in the former the interpreter can foresee `contract` as a potential future goal, in the latter, it cannot.

4.2 ASTRA(ER)

A similar experiment was run comparing ASTRA and ASTRA(ER) using the CNP benchmark. The results of this experiment are presented in figure 3. They demonstrate that ASTRA(ER) also outperforms ASTRA. This is very encouraging given that, unlike the Jason code, the two ASTRA versions are virtually identical. The main difference is the use of encapsulation of plans.

In addition to studying the performance of our new language, it is also interesting to consider the effect that encapsulated plans have on the design of larger programs. We have seen that protocols, such as CNP, benefit from the use of encapsulated plans. But, are there other areas that can benefit? One such area we have identified is queue/list processing. There are many scenarios in which an agent needs to process the items contained within a list — for example, a list could contain an agreed list of tasks to be executed by the agent. The code for such a scenario is presented in Listing 3. This

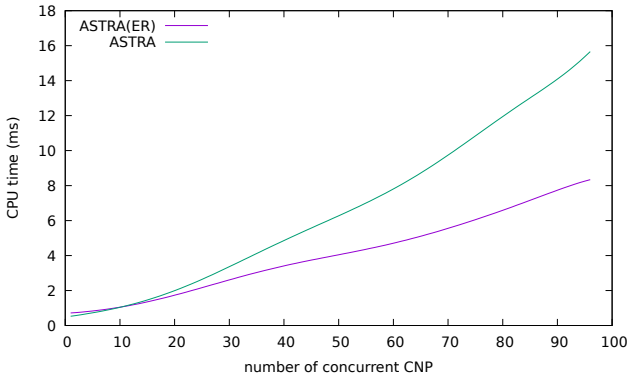

Figure 3: ASTRA(ER) performance evaluation.

listing contains two agent programs — an encapsulated plan for processing queues (`QueueProcessor`) and a second program `Main` which is a somewhat contrived example that uses this encapsulated plan to execute a set of purchases and payments.

What is interesting to note in this example is the use (and otherwise) of encapsulated plans. The `!processQueue` plan includes a plan to handle the recursive processing of the queue (including a specified delay between processing steps). However, the plan to handle the processing of an individual item is outside the goal plan. This has been done intentionally — if the plan for processing items was encapsulated within the goal rule, then it would always

Listing 3 ASTRA implementation of a Queue Processor

```
1  agent QueueProcessor {
2    module System system;
3    goal +!processQueue(list queue, int delay) {
4      body { !processQueueItems(queue); }
5      rule +!processQueueItems([funct I | list T]) {
6        !processItem(I);
7        if (list_count(T) > 0) {
8          system.sleep(delay);
9          !processQueueItems(T);
10       }
11     }
12   }
13   rule +!processQueueItem(funct I) {system.fail();}
14 }
15 agent Main extends QueueProcessor {
16   initial !performTasks([buy("apples"), pay(2)]);
17   goal +!performTasks(list events) {
18     body {
19       !processQueue(events, 3000);
20     }
21     rule +!processItem(buy(string I)) {
22       console.println("buying: "+ I);
23     }
24     rule +!processItem(pay(int amt)) {
25       console.println("paying: " + amt);
26     }
27   }
28 }
```

be given precedence over any other rules as our algorithm checks plans in the intention before it checks plans outside of the intention. This has the effect of making the behaviour immutable, in that it can never be overridden by other code. By declaring the plan outside of the goal plan, the plan becomes mutable — we can override the plan in any agent program that extends the `QueueProcessor` program.

This is precisely what the second agent program in listing 3 does. Here, we see that there is a single goal plan that declares a subgoal that can be handled by the `!processQueue` plan. However, in this case, the `!performTasks` goal plan also includes encapsulated rules for handling the specific types of task that are given in the input queue. The point here is that we are able to define a mutable templated behaviour (the `QueueProcessor`) which can be included in different programs where the expected behaviour is defined immutably (so how the `Main` program responds to the `buy(...)` and `pay(...)` tasks can never be changed). However, the program can be further extended to support additional tasks as is required.

This type of support speaks directly to long cited needs for libraries of reusable agent behaviours. While the implementation of templated behaviour was possible in AgentSpeak(L), they were reliant on hidden state and could easily be modified to work in unintended ways.

In order to better illustrate the potential of the approach, we refer the reader to a larger example that is available online. This example implements a one-shot first price auction infrastructure using a combination of the FIPA request, subscribe, and CNP protocols together with the `QueueProcessor` example shown above which is used to simulate expected interactions with the system.

5 RELATED WORK

Several implementations and extensions of the BDI model have been proposed in literature, since the PRS [17], dMARS [11] and AgentSpeak(L) [25]. Several agent programming languages and development platforms in the BDI tradition are available, besides Jason [1, 4] and ASTRA [7] used in this paper. Main examples are JADEX [24], 3APL [15] and 2APL [8], GOAL [14], Jack [32], SRI's SPARK [20], and JAM [16]. Among the most recent ones, we mention CANplan [27], an agent-oriented programming language that enhances usual BDI programming style with three distinguished features, namely declarative goals, look-ahead planning, and failure handling. All existing implementations preserve the original elements of the BDI plan model based on plans where the triggering conditions can be events about the environment and the plan body is a course of action including sub-goals and primitive actions.

As far as authors' knowledge, AgentSpeak(ER) [26] is the proposal extending the plan model in the direction discussed in this work. This paper generalises and extends the model proposed in [26], providing both a formalisation and a concrete implementation and evaluation based on two different platforms, namely Jason and ASTRA.

Devising effective programming models for integrating proactive and reactive behaviour is a general aim of Agent-Oriented Programming [30] and different approaches can be found in literature proposed for agent programming languages, not strictly based on BDI.

Finally, encapsulation is strongly related to modularity, typically impacting on how (agent) programs are organised. So this work is related to contributions in the literature focusing on improving modularity in BDI agent programming [5, 6, 9, 13, 19, 21–23, 31]. In that literature, modules are typically used as a mechanism to structure agent programs in separate parts (modules), each encapsulating cognitive components such as beliefs, goals, and plans that together model a specific functionality and can be used to handle specific situations or tasks [9]. From a software engineering point of view, modules allow a programmer to focus on those skills that are required to handle a situation [13]. In that perspective, the approach proposed in this paper improves modularity by devising coarse-grained plans encapsulating goal-oriented *and* reactive behaviour. This approach can be integrated with existing more comprehensive proposal about modularity as [19], where a module is meant to be a composable subset of the functionality of an agent, represented by a functional unit encapsulating goals, beliefs, and plans.

6 CONCLUSIONS

In this paper, we have put forward a new model for plans in BDI agent programming that provides encapsulation of both proactive and reactive behaviour. We have given algorithms for how that model is to be used in a BDI agent reasoning cycle in a way that is abstract enough to be useful for implementation in existing and future BDI platforms. That is, the algorithms are based on abstract syntax so that the approach is language independent. Furthermore, we have implemented these algorithms in two different platforms, namely Jason and ASTRA. Finally, we have reported on experimental results showing that at least in some applications our approach leads to significant efficiency compared to the traditional approach. More importantly, our examples show the elegance of our model when instantiated in particular languages and in fact some of the advantages for programming practice.

Future work includes implementing the model in other languages and further evaluating its performance. More importantly, we aim to develop more complex systems with the extended languages in order to fully assess the practical impact of our approach on agent development.

REFERENCES

[1] Rafael Bordini and Jomi Hübner. 2006. BDI Agent Programming in AgentSpeak Using Jason. In *CLIMA VI*, Francesca Toni and Paolo Torroni (Eds.). LNAI, Vol. 3900. Springer, 143–164.

[2] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni (Eds.). 2005. *Multi-Agent Programming: Languages, Platforms and Applications*. Multiagent Systems, Artificial Societies, and Simulated Organizations, Vol. 15. Springer.

[3] Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni (Eds.). 2009. *Multi-Agent Programming, Languages, Tools and Applications*. Springer. https://link.springer.com/book/10.1007/978-0-387-89299-3

[4] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldrige. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons. https://doi.org/10.1002/9780470061848

[5] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. 2006. Extending the Capability Concept for Flexible BDI Agent Modularization. In *Programming Multi-Agent Systems*, Rafael H. Bordini, Mehdi M. Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–155.

[6] Paolo Busetta, Nicholas Howden, Ralph Rönnquist, and Andrew Hodgson. 2000. Structuring BDI Agents in Functional Clusters. In *Intelligent Agents VI. Agent Theories, Architectures, and Languages: 6th International Workshop, ATAL'99, Orlando, Florida, USA, July 15-17, 1999. Proceedings*, Nicholas R. Jennings and Yves

Lespérance (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 277–289.
[7] Rem W. Collier, Sean Edward Russell, and David Lillis. 2015. Reflecting on Agent Programming with AgentSpeak(L). In *PRIMA 2015: Principles and Practice of Multi-Agent Systems - 18th International Conference, Bertinoro, Italy, 2015, Proceedings (Lecture Notes in Computer Science)*, Qingliang Chen, Paolo Torroni, Serena Villata, Jane Yung-jen Hsu, and Andrea Omicini (Eds.), Vol. 9387. Springer, 351–366.
[8] Mehdi Dastani. 2008. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems* 16, 3 (01 Jun 2008), 214–248. https://doi.org/10.1007/s10458-008-9036-y
[9] M. Dastani and B. Steunebrink. 2009. Modularity in BDI-Based Multi-agent Programming Languages. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, Vol. 2. 581–584. https://doi.org/10.1109/WI-IAT.2009.214
[10] Louise A. Dennis and Berndt Farwer. 2008. Gwendolen: A BDI Language for Verifiable Agents. In *Logic and the Simulation of Interaction and Reasoning*, Benedikt Löwe (Ed.). AISB, Aberdeen. AISB'08 Workshop.
[11] Mark D'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. 2004. The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System. *Autonomous Agents and Multi-Agent Systems* 9, 1-2 (July 2004), 5–53. https://doi.org/10.1023/B:AGNT.0000019688.11109.19
[12] Foundation for Intelligent Physical Agents. 2000. *FIPA Contract Net Interaction Protocol.* Geneva, Switzerland. http://www.fipa.org
[13] Koen Hindriks. 2008. Modules as Policy-Based Intentions: Modular Agent Programming in GOAL. In *Programming Multi-Agent Systems: 5th International Workshop, ProMAS 2007 Honolulu, HI, USA, May 15, 2007 Revised and Invited Papers*, Mehdi Dastani, Amal El Fallah Seghrouchni, Alessandro Ricci, and Michael Winikoff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–171.
[14] Koen V. Hindriks. 2009. Programming Rational Agents in GOAL. In *Multi-Agent Programming: Languages, Tools and Applications*, Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H. Bordini (Eds.). Springer US, 119–157. http://dx.doi.org/10.1007/978-0-387-89299-3_4
[15] Koen V. Hindriks, Frank S. De Boer, Wiebe Van Der Hoek, and John-Jules Ch. Meyer. 1999. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems* 2, 4 (Nov. 1999), 357–401. https://doi.org/10.1023/A:1010084620690
[16] Marcus J. Huber. 1999. JAM: A BDI-theoretic Mobile Agent Architecture. In *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS '99)*. ACM, New York, NY, USA, 236–243. https://doi.org/10.1145/301136.301202
[17] Francois F. Ingrand, Michael P. Georgeff, and Anand S. Rao. 1992. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert: Intelligent Systems and Their Applications* 7, 6 (Dec. 1992), 34–44. https://doi.org/10.1109/64.180407
[18] Brian Logan, John Thangarajah, and Neil Yorke-Smith. 2017. Progressing Intention Progression: A Call for a Goal-Plan Tree Contest. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS '17)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 768–772. http://dl.acm.org/citation.cfm?id=3091125.3091234
[19] Neil Madden and Brian Logan. 2010. Modularity and Compositionality in Jason. In *Programming Multi-Agent Systems: 7th International Workshop, ProMAS 2009, Budapest, Hungary, May 10-15, 2009. Revised Selected Papers*, Lars Braubach, Jean-Pierre Briot, and John Thangarajah (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–253.
[20] David Morley and Karen Myers. 2004. The SPARK Agent Framework. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2 (AAMAS '04)*. IEEE Computer Society, Washington, DC, USA, 714–721. http://dl.acm.org/citation.cfm?id=1018410.1018821
[21] Peter Novák and Jürgen Dix. 2006. Modular BDI Architecture. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*. ACM, New York, NY, USA, 1009–1015. https://doi.org/10.1145/1160633.1160814
[22] Ingrid Nunes. 2014. Improving the Design and Modularity of BDI Agents with Capability Relationships. In *Engineering Multi-Agent Systems: Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk (Eds.). Springer International Publishing, Cham, 58–80.
[23] Gustavo Ortiz-Hernández, Jomi Fred Hübner, Rafael H. Bordini, Alejandro Guerra-Hernández, Guillermo J. Hoyos-Rivera, and Nicandro Cruz-Ramírez. 2016. A Namespace Approach for Modularity in BDI Programming Languages. In *Engineering Multi-Agent Systems: 4th International Workshop, EMAS 2016, Singapore, Singapore, May 9-10, 2016, Revised, Selected, and Invited Papers*, Matteo Baldoni, Jörg P. Müller, Ingrid Nunes, and Rym Zalila-Wenkstern (Eds.). Springer International Publishing, Cham, 117–135.
[24] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. 2005. Jadex: A BDI Reasoning Engine. See [2], 149–174.
[25] Anand S. Rao. 1996. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings (Lecture Notes in Computer Science)*, Walter Van de Velde and John W. Perram (Eds.), Vol. 1038. Springer, 42–55.
[26] Alessandro Ricci, Rafael H. Bordini, Jomi F. Hübner, and Rem Collier. 2018. AgentSpeak(ER): An Extension of AgentSpeak(L) Improving Encapsulation and Reasoning About Goals. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '18)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2054–2056. http://dl.acm.org/citation.cfm?id=3237383.3238069
[27] Sebastian Sardina, Lavindra de Silva, and Lin Padgham. 2006. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*. ACM, New York, NY, USA, 1001–1008. https://doi.org/10.1145/1160633.1160813
[28] Sebastian Sardina and Lin Padgham. 2011. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems* 23, 1 (01 Jul 2011), 18–70.
[29] Sebastian Sardina and Lin Padgham. 2011. A BDI Agent Programming Language with Failure Handling, Declarative Goals, and Planning. *Autonomous Agents and Multi-Agent Systems* 23, 1 (July 2011), 18–70. https://doi.org/10.1007/s10458-010-9130-9
[30] Yoav Shoham. 1993. Agent-Oriented Programming. *Artif. Intell.* 60, 1 (1993), 51–92. https://doi.org/10.1016/0004-3702(93)90034-9
[31] M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Ch. Meyer, and Frank S. de Boer. 2006. Goal-oriented Modularity in Agent Programming. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*. ACM, New York, NY, USA, 1271–1278.
[32] Michael Winikoff. 2005. JACK^{TM} Intelligent Agents: An Industrial Strength Platform. See [2], 175–193.