Leveraging New Plans in AgentSpeak(PL)

Felipe Meneguzzi and Michael Luck

King's College London
Department of Computer Science
Strand, London WC2R 2LS, UK
{felipe.meneguzzi, michael.luck}@kcl.ac.uk

Abstract. In order to facilitate the development of agent-based software, several agent programming languages and architectures, have been created. Plans in these architectures are often self-contained procedures with an associated *triggering event* and *a context condition*, while any further information about the consequences of executing a plan is absent. However, agents designed using such an approach have limited flexibility at runtime, and rely on the designer's ability to foresee all relevant situations an agent might have to handle. In order to overcome this limitation, we have created AgentSpeak(PL), an interpreter capable of performing state-space planning to generate new high-level plans. As the planning module creates new plans, the plan library is expanded, improving performance over time. However, for new plans to be useful in the long run, it is critical that the *context condition* associated with new plans is carefully generated. In this paper we describe a plan reuse technique aimed at improving an agent's runtime performance by deriving optimal context conditions for new plans, allowing an agent to reuse generated plans as much as possible.

1 Introduction

Software based on autonomous agents is often advocated as a solution to addressing highly dynamic environments in which human intervention is impractical or impossible. In order to facilitate the development of such agent-based software, several agent programming languages, as well as associated agent architectures, have been created. So far, however, for reasons of efficiency, the set of practical agent architectures developed has mainly focused on providing a *plan execution* framework for a plan library defined at design time [1, 2]. Plans in these architectures are often self-contained procedures with an associated *triggering event*, while any additional information about the consequences of executing a plan is absent. For example, PRS [3] and its successors [4, 5] provide concrete agents which, while efficient, are noticeably inflexible in handling anything not foreseen at design-time.

Traditional BDI agents [2] are designed using a procedural approach, which requires a designer to create detailed procedural plans for every relevant situation in which an agent may find itself prior to deployment. Situations in which plans must be executed are encoded in a plan header in two parts: a triggering event, identifying the moment when a plan may be necessary; and a context condition describing the pre-requisites for the plan to be applicable, as shown in Figure 1. Both the triggering event and the

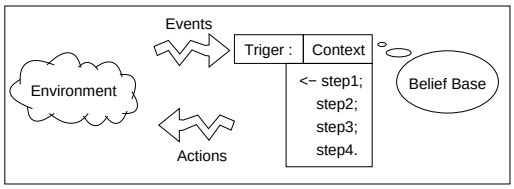

Fig. 1: AgentSpeak(L) plan and dynamics.

context condition are defined statically at design time, and ensure that a plan can execute successfully when it is required. However, agents designed using such an approach have limited flexibility at runtime, and rely on the designer's ability to foresee all relevant situations an agent might have to handle.

In order to overcome this limitation, we have created AgentSpeak(PL) [6], an interpreter capable of generating new high-level plans when no suitable plans exist in the plan library. These high-level plans are created by sequencing existing lower-level plans from the plan library, from which key information about their declarative preconditions and consequences is extracted. AgentSpeak(PL) uses state-space planning to create new plans, and since state-space planners are inherently declarative, AgentSpeak(PL) is able to reason about declarative goals and create plans which, when executed, ensure that a certain world-state is true. The approach taken in AgentSpeak(PL) consists of evaluating the consequences of procedural plans in terms of belief additions and deletions and converting these plans into a STRIPS-like representation, which can then be supplied to a classical planner [7] along with the current belief base and the desired goal state. In this setting, STRIPS operators are essentially an analogue for lower-level AgentSpeak(L) plans and, therefore, plans generated by the planning module represent high-level AgentSpeak(L) plans, which consist entirely of lower-level plan invocations. As the planning module creates new plans, the plan library is expanded, improving performance over time.

However, for new plans to be useful for an agent in the long run, it is critical that the *context condition* associated with new plans is simple enough so that plans can be executed whenever they accomplish their goals, and restrictive enough, so that plans do not execute in situations in which they would fail. As an example, suppose that a certain agent has a car and a motorcycle available to move it from home to work, and an action to drive each one of these vehicles having a precondition that the vehicle being used must have enough fuel for the journey. Furthermore, suppose that at one point in time this agent generated a plan to drive its car to work, while believing that both the motorcycle and the car had enough fuel. The precondition of the high-level plan involving the car surely must contain the belief regarding the car's fuel level, but not the motorcycle's, as it is irrelevant to that plan.

While previous work [6] focused on the integration of the interpreter with the planner through a translation process, in this paper we focus on specific aspects of adding new plans to the plan library. The key contribution is an improvement in an agent's runtime performance by deriving optimal context conditions for new plans, allowing an agent to reuse generated plans as much as possible. We evaluate the resulting sys-

tem against a naive strategy of plan reuse, as well as a similar agent designed using AgentSpeak(L), in order to demonstrate the efficiency of our approach. Although we use AgentSpeak(L) as a demonstration platform, our approach can also be applied to other planning-capable agent architectures.

This paper is organised as follows: Section 2 reviews previous work on AgentSpeak(PL), providing the necessary background for this paper; Section 3 describes our plan reuse strategy, including an algorithm for context generation; Section 4 reports on the experiments performed using an implementation of our strategy and its results for a production cell scenario; Section 5 provides a brief overview of recent related work in comparison to ours; finally, Section 6 draws conclusions from our results and proposes future research based on them.

2 AgentSpeak(PL)

AgentSpeak(PL) [6] is an extended AgentSpeak(L) interpreter that uses a planning component to reason about declarative goals. In this section we briefly describe both the original AgentSpeak(L) interpreter and language, and the extensions provided in AgentSpeak(PL).

2.1 AgentSpeak(L)

AgentSpeak(L) [2] is an agent language, as well as an abstract interpreter for the language, that follows the *beliefs, desires and intentions* (BDI) model of practical reasoning [8]. In simple terms, a BDI agent tries to realise the *desires* it *believes* are possible by committing to carrying out certain courses of action through *intentions*. The language of AgentSpeak(L) allows the definition of *reactive procedural plans*, so that plans are defined in terms of an event to which an agent should react to by executing a sequence of steps (*i.e.* a procedure). Plan execution is further constrained by the *context* in which these plans are relevant. Here, a plan is executed under the assumption that some implicit goal is being accomplished by that plan at that particular moment.

The control cycle of an AgentSpeak(L) interpreter is driven by events relating to either new beliefs (including perceptions) and new goals. These events are used as triggering conditions for the adoption of plans, so that adding an achievement goal means that an agent desires to fulfil the goal, and plans whose triggering condition includes that goal (*i.e.* are *relevant* to the goal) should lead to that goal being achieved. Moreover, a plan includes a logical *context* condition that specifies when the plan is *applicable* (*i.e.* possible to be executed) in any given situation. Whenever a goal addition event is generated (as a result of the currently selected plan having subgoals), the interpreter searches the set of relevant plans for applicable plans; if one (or more) such plan is found, it is pushed onto an intention structure for execution. Elements in the intention structure are popped and handled by the interpreter. If the element is an action it is executed, while if the element is a goal, a new plan for that goal is added into the intention structure and processed. During this process, failures may take place either in the execution of actions, or during the processing of subplans. When such a failure takes place, the plan that is currently being processed also fails. Thus, if a plan selected for the achievement

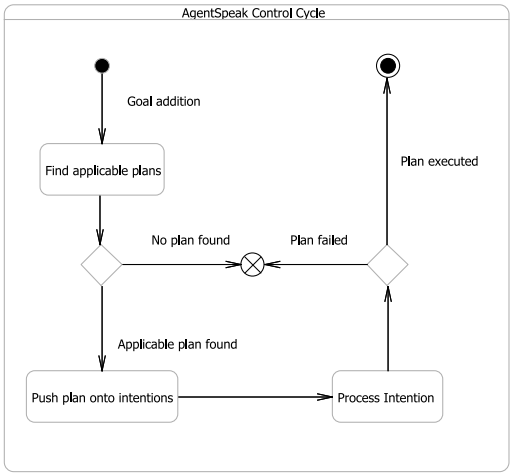

Fig. 2: AgentSpeak(L) reasoning cycle.

of a given goal fails, the default behaviour of an AgentSpeak(L) agent is to conclude that the goal that caused the plan to be adopted is not achievable. This control cycle is illustrated in the diagram of Figure 2,[1] and strongly couples plan execution to goal achievement.

In order to better understand the relationship between the control cycle and the plan library, it is necessary to introduce the notation of AgentSpeak(L) plans. The events on an agent's data structures that can trigger the adoption of plans consist of additions and deletions of goals and beliefs, and are represented by the plus ($+$) and minus ($-$) sign respectively. Goals are distinguished into *test goals* and *achievement goals*, denoted by a preceding question mark ($?$), or an exclamation mark ($!$), respectively. For example, the addition of a goal to achieve g is represented by $+!g$, whereas the addition of a goal to test the truth value of a belief b is represented by $+?b$. Belief additions and deletions arising as the agent perceives the environment are outside its control, while goal additions and deletions and some belief modifications only arise as part of the execution of an agent's plans. Plans in AgentSpeak(L) are represented by a header comprising a triggering condition and a context, as well as a body describing the steps the agent takes when a plan is selected for execution, as illustrated in Figure 1. Thus, if e is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n$ are goals or actions, then $e : b_1 \& \ldots \& b_m \leftarrow h_1; \ldots; h_n.$ is a plan. As an example, consider a plan associated with the triggering event `!move(O,A,B)` corresponding to an achievement goal to move an object O from A to B, where:

- e is `!move(O,A,B)`;
- `at(O,A)` and **`not`** `at(O,B)` are belief literals; and
- `-at(O,A)` and `+at(O,B)` are two steps in the plan body, consisting of information about belief additions and deletions.

The plan is then as follows:

[1] For a full description of AgentSpeak(L), refer to d'Inverno *et al.*[9]

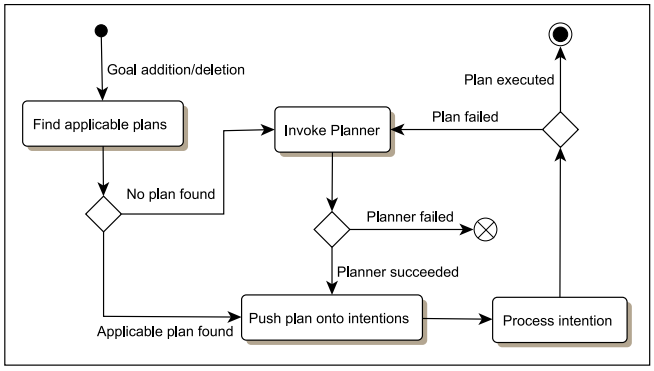

Fig. 3: AgentSpeak(PL) reasoning cycle.

```
+!move(O,A,B) : at(O,A) & not at(O,B)
    <- -at(O,A);
       +at(O,B).
```

When this plan is executed, it should result in the agent believing O is no longer in position A, and then believing it is in position B. For an agent to rationally want to move O from A to B, it must believe O is at position A and not already at position B.

2.2 Planning in AgentSpeak(PL)

In order to overcome the limitations of traditional AgentSpeak(L) programming in terms of dynamic plan generation and declarative goal representation, previous work has introduced AgentSpeak(PL) [6], which is an extended AgentSpeak(L) interpreter coupled with a planning module able to perform STRIPS-like planning. The agent interpreter communicates with the planning module through a translation process that relies on the similarities between AgentSpeak(L) plans and STRIPS operators. This is possible because both formalisms describe world modification functions that can be applied if certain preconditions hold, resulting in changes to the world-state.

The planning action In addition to the traditional way of encoding goals for an AgentSpeak(L) agent implicitly as triggering events consisting of achievement goals ($!goal$), AgentSpeak(PL) allows desires including multiple beliefs $(b_1, \ldots, b_n)$ describing a desired world-state in the form $goal\_conj([b_1, \ldots, b_n])$. An agent desire description thus consists of a conjunction of beliefs the agent wishes to be true simultaneously at a given point in time. The execution of the planning component is triggered by an event $+goal\_conj([b_1, \ldots, b_n])$ as shown in Table 1.

In this approach, planning in AgentSpeak is introduced through a special *planning action*, denoted $plan(G)$, where G is a conjunction of desired goals. This action is bound to an implementation of a planning module, and allows all of the process regarding the conversion between formalisms to be encapsulated in the action implementation, making it completely transparent to the remainder of the interpreter. Note that there are

$$+goal\_conj(Goals) : true \leftarrow plan(Goals).$$

Listing 1: Planner invocation plan.

two different steps in invoking the planning action: the declarative goal, represented by the $+goal\_conj(Goals)$ event; and the planner invocation action $plan$, which may occur as a consequence of adopting a declarative goal.

Whenever an agent needs to achieve a goal that involves planning, it uses a special planning action that converts the low-level procedural plans of AgentSpeak(L) into STRIPS operators and invokes the planning module. If the planner succeeds in finding a plan, it is converted back into a high-level AgentSpeak(L) plan and is added to the intention structure for execution, as illustrated in Figure 3. This conversion process is detailed in Section 2.2. If the newly created plan fails, the planner may again be invoked to try to find another plan to achieve the desired state of affairs, taking into consideration any changes in the beliefs.

Note that the planning action is included in a standard AgentSpeak plan with the same triggering condition as the plans generated by it. Moreover, new plans are always added to the plan library *before* the plan that executes the planning action. With this arrangement, previously-created plans are consulted first when the interpreter searches for relevant plans, hence having higher priority for execution. If no such plan is found to be applicable, the plan containing the special planning action is invoked as the last remaining option. An important limitation imposed by our current implementation is that, since goal conjunctions are represented as lists, different goal orderings correspond to different declarative goals and, therefore, a goal to achieve $[a, b]$ is not seen as the same goal to achieve $[b, a]$.

Translating AgentSpeak into STRIPS Once the need for planning is detected, the plan in Table 1 is invoked so that the agent can tap into a planning component. The process of linking an agent to a propositional planning algorithm includes converting an AgentSpeak plan library into propositional planning operators, declarative goals into goal-state specifications, and the agent beliefs into the initial-state specification for a planning problem. After the planner yields a solution, the ensuing STRIPS plan is translated into an AgentSpeak plan in which the operators resulting from the planning become subgoals. That is, the execution of each operator listed in the STRIPS plan is analogous to the insertion of the AgentSpeak plan that corresponded to that operator when the STRIPS problem was created.

In classical STRIPS notation, operators have four components: an identifier, a set of preconditions, a set of predicates to be added (add), and a set of predicates to be deleted (del). For example, the same `move` operator can be represented in STRIPS following the correspondence illustrated in Figure 4, in which AgentSpeak(PL) converts the invocation condition into a STRIPS operator header, a context condition into an operator precondition, and the plan body is used to derive add and delete lists.

A relationship between these two definitions is not hard to establish, and AgentSpeak(PL) defines the following algorithm for converting AgentSpeak (low-level) plans

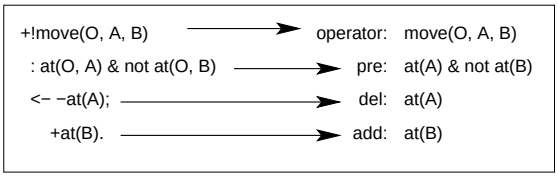

Fig. 4: AgentSpeak plan versus STRIPS operator.

$$+goal\_conj(Goals) : true \leftarrow !op_1; \ldots; !op_n.$$

Listing 2: AgentSpeak plan generated from a STRIPS plan.

into STRIPS operators. Let e be a triggering event, $b_1 \& \ldots \& b_m$ a conjunction of belief literals representing a plan's context, $a_1, \ldots, a_n$ be belief addition actions, and $d_1, \ldots, d_o$ be belief deletion actions within a plan's body. All of these elements can be represented in a single AgentSpeak plan. Moreover let $opname$ be the operator name and parameters, pre be the preconditions of the operator, add the predicate addition list, and del the predicate deletion list. Mapping an AgentSpeak plan into STRIPS operators is then accomplished as follows:

1. $opname = e$
2. $pre = b_1 \& \ldots \& b_m$
3. $add = a_1, \ldots, a_n$
4. $del = d_1, \ldots, d_o$

Above, we have introduced the representation of a conjunction of desired goals as the predicate $goal\_conj([b_1, \ldots, b_n])$. The list $[b_1, \ldots, b_n]$ of desires is directly translated into the goal state of a STRIPS problem. Moreover, the initial state specification for a STRIPS problem is generated directly from the agent's belief database.

Executing generated plans The STRIPS problem generated from the set of operators, initial state and goal state is then processed by a propositional planner. If the planner fails to generate a propositional plan for that conjunction of literals, the plan in Table 1 fails immediately and the goal is deemed unachievable, otherwise the resulting propositional plan is converted into an AgentSpeak plan and added to the intention structure. A propositional plan from a STRIPS planner is in the form of a sequence $op_1, \ldots, op_n$ of operator names and instantiated parameters. AgentSpeak(PL) creates a new plan as in Table 2, where $goal\_conj(Goals)$ is the event that caused the planner to be invoked.

Immediately after adding the new plan to the plan library, the event $goal\_conj(Goals)$ is reposted to the agent's intention structure, causing the generated plan to be executed. The abstract language of AgentSpeak(L) does not include constructs for plan library modification, but this type of functionality is generally accomplished through internal actions by many interpreters, including *Jason* [10]. As a consequence, there is no commonly agreed semantics for the addition of new plans into

an agent's plan library. The method we use is that of the *Jason* interpreter, which consists of inserting the new plan either at the beginning of the plan library or at its end. Plans generated in this fashion are admittedly simple, and in order for an agent to take full advantage of the planning module, we need to consider how plans should be added to the plan library for future reference.

2.3 Limited plan reusability

However, the addition of the new plan to the intention structure raises the problem of how newly formed plans can be integrated into the agent's existing plan library, or indeed if they should be integrated into the plan library at all. Modifying the plan library at runtime through the addition of new plans effectively changes agent behaviour in at least two ways: first, new plans may cause undesired interactions with the plans that are already part of the plan library, possibly jeopardising the agent's viability in the long term; and second, adding a large number of plans with the same invocation condition may impair the agent's ability to respond in adequate time.

In order for a plan to be usefully added to the plan library, therefore, the *context* in which this plan is relevant must be carefully described. If the context is too restrictive, for example by using the entire belief base at the time of planning, the inclusion of a number of irrelevant beliefs will severely limit the future applicability of the new plan. On the other hand if the context is minimised to only the preconditions of the first operator, the plan may fail later on due to the requirements of subsequent operators. In consequence, an algorithm that generates the minimum context condition necessary for a newly generated plan to be reused usefully is required.

3 Leveraging new plans

In order to address the need for a minimum context condition for newly created plans, we have developed an algorithm to extract the minimum necessary context condition from a planner-generated plan that ensures that if the context is true when the plan is adopted, it will succeed if no external interference takes place.

3.1 Data structure

We base our context-generation algorithm on a modified version of the *planning graph* data structure from Graphplan [11], which is a graph-based planning algorithm. The properties of this method of plan representation are key to our algorithm and, therefore, before describing the algorithm, we introduce the planning graph. Since a plan is composed of temporally ordered actions, and these actions alter propositions in the intermediate world states, graph levels are divided into alternating proposition and action levels, making it a directed and levelled graph. A graphical representation of one such graph is shown in Figure 5, in which oval shapes denote propositions and boxes denote actions. Proposition levels are composed of proposition nodes labelled with propositions, and are connected to the actions in the subsequent action level through precondition arcs. Here, action nodes are labelled with operators and are connected to the nodes in the

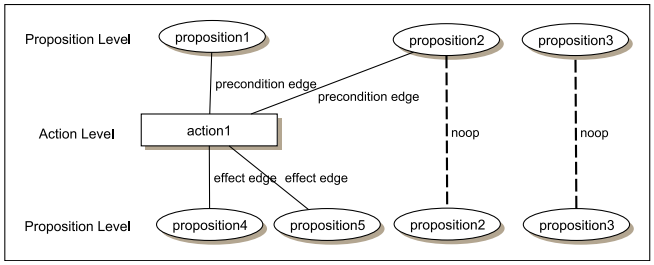

Fig. 5: A planning graph example.

subsequent proposition nodes by effect arcs, and both added and deleted propositions are possible effects of an operator.

Every proposition level denotes literals that are possibly true at a given moment, so that the first proposition level represents the literals that are possibly true at time t_1 (the initial time), the next proposition level represents the literals that are possibly true at time t_2 and so on. Similarly, action levels denote operators that can be executed at a given moment in time in such a way that the first action level represents the operators that may be executed at time t_1, the second action level represents the operators that may be executed at time t_2 and so on.

The process of building a graph in Graphplan consists of initialising it with a proposition level containing the initial state of the planning problem, and adding all actions that have their entire set of preconditions present in that proposition level. New proposition levels are then created, including all the effects of the preceding action level. In order to guarantee a static frame for all actions in the graph (that is, to ensure propositions not affected by plan operators remain unchanged between points in time), *no operation* (or *noop*) edges are inserted between propositions to represent the possibility that these propositions are not changed between two proposition levels (*i.e.* points in time). The planning graph used in Graphplan has a number of other characteristics that we do not explain in this paper because they are not relevant to our algorithm, but are discussed in [11].

3.2 Generating context information

Intuitively, the preconditions of any given plan step must have either been made true during the execution of previous plan steps or must have been true from the start of the plan. Therefore, the minimum context condition for any generated plan must specify the preconditions of the first operator, plus the preconditions of any subsequent operators that are not included in the effects of previous operators. We consider this process in more detail in Algorithm 1, which describes the generation of such a context condition.

The algorithm initially builds a planning graph populated with the actions of the plan we wish to create a context for, as well as the preconditions and effects of these actions, with edges connecting actions to their preconditions in the previous level, and their effects in the subsequent level. Once the initial graph is generated, proposition levels are iterated backwards and, for each proposition that is connected with a precon-

Algorithm 1 Propagation of preconditions.

Require: Plan $\Delta = \{a_1, \ldots, a_n\}$, with n steps
Require: Action descriptions $\mathbf{O} = \{\langle a_1, Pre_1, Post_1\rangle, \langle a_n, Pre_n, Post_n\rangle\}$
create a proposition level P_0 with no propositions;
for all $a_i \in \Delta$ **do**
  create an action level A_i containing a node a_i;
  add the preconditions of a_i to proposition level P_{i-1};
  connect all $p \in P_i$ to a_{i-1} with precondition edges;
  create a proposition level P_i containing the effects of a_i;
  connect all $p \in P_i$ to a_i with effect edges;
end for
for $i = n$ to 1 **do**
  for all $p \in P_{i-1}$ **do**
    if p is not connected to any node in level A_i **then**
      create an action $noop(p)$ in level A_i;
      connect $noop(p)$ to p through an effect edge;
      if $p \notin P_i$ **then**
        create a node p in P_i;
      end if
      connect p to $noop(p)$ with a precondition edge;
    end if
  end for
end for
return P_0

dition edge to a subsequent action level and not connected with an effect edge to the previous action level, a new *noop* action is created, allowing a proposition to be propagated to the previous proposition level. As the graph is traversed, propositions that are required at one action level are created at the preceding proposition levels until they are either connected to an original action of the plan, or they are propagated through *noop* actions, ensuring that the first proposition level contains all of the preconditions that did not result from the actions in the plan.

In terms of computational effort, this algorithm has similar complexity to the graph expansion phase of Graphplan, which has polynomial complexity [7] in the size of the planning problem for both the size of the graph and the time required to build it. If a plan has m distinct steps, and n distinct propositions, the graph our algorithm creates will have at most $((2 * n) + 1) * m$ nodes, one node for each action and all possible *noop*s at each graph level, plus all possible propositions at each proposition level, indicating that the size and time complexity of our algorithm is in the low polynomial scale. Regarding the correctness of the algorithm and its termination guarantee, since the graph building part of the algorithm is a subset of Graphplan, for which a proof of completeness and termination exists, and the rest of the algorithm is an iteration in a directed acyclic graph, it is trivial to show that the algorithm does terminate for any input.

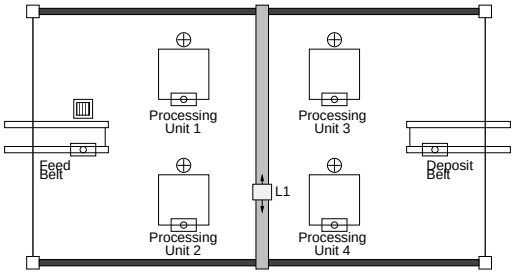

Fig. 6: Diagram of the production cell.

3.3 A production cell example

To illustrate how our algorithm derives a context condition, we introduce a production cell scenario, shown in Figure 6, and consisting of a production cell composed of four processing units (`u1`, `u2`, `u3` and `u4`) and two conveyor belts controlled by our agent. Parts enter the production cell through a *feed belt*, and are moved by the agent to different *processing units*, depending on the type of part being processed. Once a part has been processed at the appropriate processing units, it is moved to the *deposit belt* to be shipped. Even though there is no particular order specified for the processing of parts, the order in which they are specified is generally followed. We consider three different types of part for processing, in the following processing units: *i)* type one must be processed by processing units 1, 2 and 3; *ii)* type two must be processed by processing units 2 and 4; and *iii)* type three must be processed by processing units 1 and 3.

In addition, we assume two operations are available in this scenario, summarised in Table 1.[2] The first operator, `move(P,A,B)`, moves a part from one device to another, requiring the part to be over the initial device and the target device to be empty, and causing the initial device to become empty, the part to be over the target device and the deletion of the preconditions (note that while we represent explicitly negated propositions in the graph, we do not require the world to be described with explicitly negated conditions). The second operator, `process(P,A)`, causes a processing unit to process a part located over it, requiring `over(P,A)` and causing `processed(P)`.

Operator	Preconditions	Effects
`move(P,A,B)`	`empty(B)`	`~empty(B)`
	`over(P,A)`	`~over(P,A)`
		`over(P,B)`
		`empty(A)`
`process(P,A)`	`over(P,A)`	`processed(P)`

Table 1: Operations in the production cell scenario.

[2] We use a Prolog-like notation, with variable names starting in uppercase and constants in lowercase.

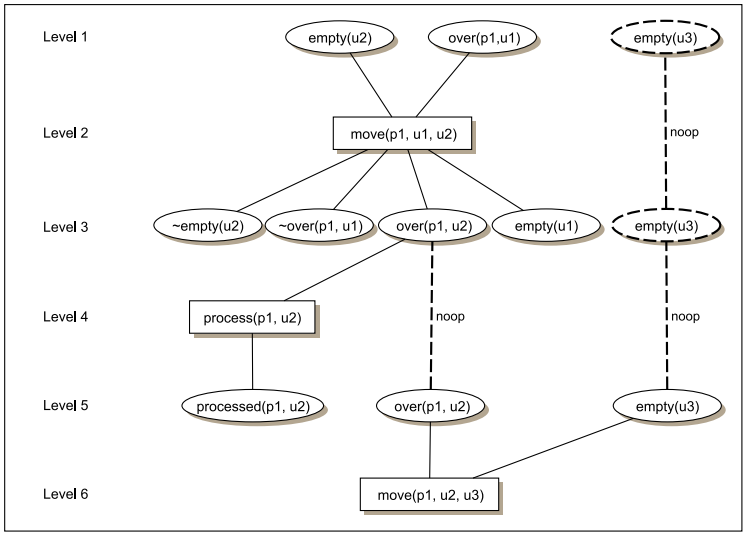

Fig. 7: A planning graph used in context extraction.

Now let us consider in more detail the process of generating the context for this example. Nodes in an action level are connected to nodes in a proposition level either through precondition edges, denoting that a proposition is a precondition of a given action, or through effect edges, denoting that a proposition is an effect of a given action. In the example of Figure 7, the operator `process(p1,u2)` in Level 4 is connected by a precondition edge to the proposition `over(p1,u2)` in Level 3, and by precondition edges to the proposition `processed(p1,u2)` in Level 5. Besides the actions included in the planning problem, the planning graph includes *noop* (or maintenance) actions, which connect identical propositions between adjacent proposition levels representing that their truth values remain unchanged between plan steps, an example of which can be seen connecting the proposition `empty(u3)` from Levels 1 to 5.

Figure 7 shows the graph generated in the process of deriving the context for a plan composed of three actions: `move(p1,u1,u2)`, `process(p1,u2)` and `move(p1,u2,u3)`. The initial graph created by our algorithm contains no maintenance actions and no instances of `empty(u3)` in Levels 1 and 3. Then, while iterating the graph backwards, the algorithm detects that none of the preconditions of `move(p1,u2,u3)` were caused by the immediately preceding action, and adds a *noop* connecting the instances of `over(p1,u3)` in Levels 3 and 5. Furthermore, it creates instances of `empty(u3)` in Levels 1 and 3, connecting them with maintenance operators in Levels 2 and 4, thus propagating `empty(u3)` to the initial plan level. Since no action in the plan resulted in `empty(u3)` being true, this must have been true before the plan was adopted to make the last action possible. These additions to the planning graph can be seen in Figure 7 as the dashed oval shapes and lines.

4 Experiments and Results

Traditional AgentSpeak(L) agents require a plan library containing plans for every conceivable situation an agent might find itself in, since no plans can be created at runtime to deal with unexpected events. Therefore, the ability to generate new plans at runtime both increases an agent's flexibility and eases agent development. On the other hand, state space planners are complex, and a decrease in runtime performance is expected over standard AgentSpeak(L). With the addition of an effective plan reuse strategy, however, the time spent in the planning process can be mitigated over time, since new plans will have the same runtime efficiency as traditional AgentSpeak(L).

Our prototypes were implemented using modified versions of Jason [10], a Java-based AgentSpeak(L) interpreter with a few additional constructs such as plan annotations and plan failure handling. Experiments with traditional AgentSpeak(L) agents were conducted in an unmodified Jason interpreter, whereas planning agents were created using the open-source implementation of AgentSpeak(PL) [6], unmodified for experiments without plan reuse; and extended with our algorithm for context generation.

The experiment consists of simulating the arrival of parts of three types in three production cells, one controlled by a traditional AgentSpeak(L) agent (AS), another controlled by a *naive* version of AgentSpeak(PL) (NaiveAS) that does not reuse plans and one controlled by the *complete* AgentSpeak(PL) (ASPL) capable of reusing plans. Here, whenever a new part arrives for processing at the cell controlled by NaiveAS, the full planning process is invoked to generate a new plan, regardless of previous instances of the same problem having been considered in the past. The time spent planning and achieving the final processing of every part is measured for each agent for an increasing number of parts, ranging from 10 to 100 in 10 part increments.

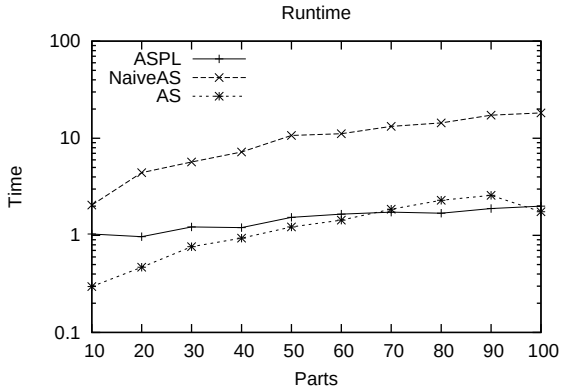

Fig. 8: Running times for the Production Cell scenario.

The results of this experiment can be seen in the graph of Figure 8, which shows that, though NaiveAS takes significantly more time to perform its reasoning cycle, this overhead is constant. Now, when the plan reuse strategy is used by ASPL, runtime

	AS	ASPL
# plans	12	7

Table 2: Plan library size comparison.

performance improves considerably, approaching that of AS. With three different part types, the number of possible world configurations at the time of planning is limited, and most of the planning effort occurs at the beginning of the agent execution. As more parts of the same type are introduced in the production cell, the plans generated previously are invoked rather than the planning module, *amortising* the cost of the initial planning. Evidence of this effect is provided by the ASPL curve approaching that of AS as the number of total parts increases. Moreover, since the plans generated through planning are a linear sequence of actions, which do not rely on the tests distributed throughout a branching structure of plans in the plan library, they are inherently faster to be executed than the equivalent AS representation, surpassing it in the long term.

It is important to note that, although ASPL can create plans for situations in which AS would fail, we have avoided using these problems in our benchmark, focusing only on runtime, by considering an AS agent with plans for all situations possible during testing. By relying on a planning approach, we also diminish the size of the agent specification, since we no longer need to create a procedural plan to cope with every world configuration relevant to the accomplishment of an individual plan. The numbers of plans necessary in the (initial) plan libraries are shown in Table 2.

5 Related Work

Work on using planning modules to augment existing architectures has been conducted by several researchers, such as in Propice-Plan [12] and JADEX [13, 14]. These efforts provide some insight into many practical issues that may arise from the integration of BDI with AI planners, such as how to modify a planning algorithm to cope with changes in the initial state during planning [12], and how to cope with conflicts in concurrently executing plans [13].

Propice-Plan [12] is a PRS-based system that includes planning capabilities through a modified version of the IPP planner [15]. It includes refinements to allow an agent to anticipate alternative execution paths for its plans, as well as the ability to update the state of the planning process in order to cope with a highly dynamic world. Propice-Plan is similar in principle to the architecture described here, but it differs in two key aspects: its reliance on a modified PRS description formalism for agents, and its reliance on a tailor-made planner implementation, limiting the choice of planners to be used in tandem with the agent interpreter.

The work of Walczak *et al.*[13] is a recent approach to merging BDI reasoning with planning capabilities, and is based on a continuous planning and execution framework implemented in the JADEX agent framework [16]. The system uses a modified HTN state-based planner with domain-specific information to select the actions to achieve goals or refine goals in an agent's agenda. The emphasis in this system is on performance and reaction time rather than generality, since JADEX uses a Java-like representation for the agent's data structures, such as goals and actions. Admittedly, an HTN

planner could be used to generate new plans following a task-oriented approach (and hence not for declarative goals), but this is not what is accomplished in JADEX.

Considering the many similarities between BDI programming languages and HTN planning, Sardina *et al.*[14] formally define how HTN planners can be integrated into a BDI architecture. Sardina shows that the HTN process of systematically substituting higher-level goal tasks until concrete actions are derived is analogous to the way in which a PRS-based interpreter pushes new plans onto an intention structure, replacing an achievement goal with an instantiated plan. Taking advantage of this almost direct correspondence, an HTN planner is used to add *lookahead* capabilities to an agent, allowing it to optimise plan selection and maximise an agent's chance of successfully achieving goals. By verifying beforehand the selection of plans for achieving subgoals, the agent minimises the chance of failure as a result of poor plan selection.

The idea of analysing one formalism to derive planning-like pre and post conditions has been attempted previously in the context of web service composition through planning. Initial efforts by McIlraith and Fadel [17] at a theoretical level, involved converting web services described by hand using Golog into PDDL and ADL. However, this lacked generality due to its heavy reliance on human intervention in the process, preventing it from being used in a completely automated fashion, as is needed by our work. Later, this idea was refined by Pistore *et al.*[18], converting web services defined in BPEL4WS into PDDL, allowing for automation. However, BPEL is much more complex than AgentSpeak, and understandably the conversion algorithm has polynomial complexity, though on the *exponential* scale. In this respect, our approach compares favourably by having non-exponential polynomial complexity.

6 Conclusions

In this paper we have described a plan reuse strategy for AgentSpeak(PL), a planning-capable extension of AgentSpeak(L), able to reason about declarative goals. This strategy is based on the generation of the simplest context information necessary for newly created plans to be successful when responding to the same event that triggered the agent to plan. Since planning is a computationally intensive task, being able to effectively reuse previously generated plans offsets time spent on the planning process, as new plans are as efficient as AgentSpeak(L) procedural plans. This bridges the performance gap between traditional AgentSpeak(L) agents and declarative goal-based agents developed in AgentSpeak(PL), which is the primary contribution of this paper.

The strategy used in the generation of context information is based on the generation of a greatly simplified planning graph analogous to the graph used in Graphplan [11]. Building this graph has low polynomial complexity in both time and space, and therefore the performance overhead imposed on the planning process is negligible. Moreover, we believe that the process used to generate initial context information can be expanded to allow reasoning about interference from concurrent plans, and we intend to explore this possibility as future work.

Acknowledgments. The first author is supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) of the Brazilian Ministry of Education.

References

1. Meneguzzi, F., Zorzo, A.F., da Costa Móra, M., Luck, M.: Incorporating planning into BDI agents. Scalable Computing: Practice and Experience **8** (2007) 15–28
2. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In de Velde, W.V., Perram, J.W., eds.: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World. Volume 1038 of LNCS. Springer (1996) 42–55
3. Rao, A.S., Georgeff, M.P.: BDI-agents: from theory to practice. In: Proceedings of the First International Conference on Multiagent Systems, San Francisco (1995) 312–319
4. d'Inverno, M., Luck, M., Georgeff, M., Kinny, D., Wooldridge, M.: 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
5. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the golden fleece of agent-oriented programming. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: Multi-Agent Programming: Languages, Platforms and Applications. Springer (2005) 3–37
6. Meneguzzi, F., Luck, M.: Composing high-level plans for declarative agent programming. In: Proceedings of the Fifth Workshop on Declarative Agent Languages. (2007) 115–130
7. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Elsevier (2004)
8. Bratman, M.E.: Intention, Plans and Practical Reason. Harvard University Press, Cambridge, MA (1987)
9. d'Inverno, M., Luck, M.: Engineering AgentSpeak(L): A formal computational model. Journal of Logic and Computation **8**(3) (1998) 233–260
10. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley (2007)
11. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial Intelligence **90**(1-2) (1997) 281–300
12. Ingrand, F., Despouys, O.: Extending procedural reasoning toward robot actions planning. In: Proceedings of the 2001 IEEE International Conference on Robotics and Automation, Seoul, Korea (2001) 9–10
13. Walczak, A., Braubach, L., Pokahr, A., Lamersdorf, W.: Augmenting BDI Agents with Deliberative Planning Techniques. In: Proceedings of the Fifth International Workshop on Programming Multiagent Systems. (2006)
14. Sardina, S., de Silva, L., Padgham, L.: Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems. (2006) 1001–1008
15. Köhler, J.: Solving complex planning tasks through extraction of subproblems. In Simmons, R., Veloso, M., Smith, S., eds.: Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, Pittsburg, AAAI Press (1998) 62–69
16. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: Multi-Agent Programming: Languages, Platforms and Applications. Springer (2005) 149–174
17. McIlraith, S.A., Fadel, R.: Planning with complex actions. In Benferhat, S., Giunchiglia, E., eds.: Proceedings of the 9th International Workshop on Non-Monotonic Reasoning. (2002) 356–364
18. Pistore, M., Marconi, A., Bertoli, P., Traverso, P.: Automated composition of web services by planning at the knowledge level. In Kaelbling, L.P., Saffiotti, A., eds.: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence. (2005) 1252–1259